ROBSON JOÃO PADILHA DA LUZ

UM ALGORITMO PARA A EVOLUÇÃO INCREMENTAL DE ESQUEMAS PARA XML

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Martin A. Musicante

CURITIBA

2007

ROBSON JOÃO PADILHA DA LUZ

UM ALGORITMO PARA A EVOLUÇÃO INCREMENTAL DE ESQUEMAS PARA XML

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Martin A. Musicante

CURITIBA

2007

AGRADECIMENTOS

Agradeço a todas as pessoas, professores e amigos, que de uma forma ou outra contribuíram para a realização deste trabalho, e que me ajudaram em todo o meu tempo dedicado ao mestrado.

De maneira especial agradeço ao Professor Martin A. Musicante que sempre me apoiou, me orientando, da melhor forma possível, para a realização deste trabalho.

À minha família e à Rosane que sempre me apoiaram na realização deste mestrado, e principalmente a Deus, o meu maior inspirador.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

A utilização de documentos XML para representação, armazenamento, e transporte de dados vem aumentando a cada dia. Muitas organizações utilizam documentos XML para armazenar suas informações, onde geralmente a quantidade de documentos necessários para este armazenamento é relativamente grande. Os esquemas XML ajudam a organizar estes documentos, estabelecendo restrições de como eles devem ser formados. Alterações nos documentos XML podem ser necessárias para atender novas necessidades. Estas alterações podem invalidar os esquemas correspondentes aos documentos XML alterados. Então existe a necessidade de realizar alterações nestes esquemas, sem invalidar os documentos XML reconhecidos anteriormente. Dada a necessidade de trabalhar de forma automatizada com essas alterações nos esquemas, este trabalho propõe um algoritmo de evolução incremental de esquemas para XML. A evolução de esquemas é incremental no sentido de ser conservativa às características presentes no esquema antigo. O processo de evolução incremental em um esquema XML D é realizado a partir de uma lista de modificações L realizadas em um documento XML X que é reconhecido por D. Para cada modificação μ em L, o esquema D é evoluído para que aceite os documentos que estejam de acordo com esta modificação. Após o processo de evolução sobre cada modificação em L, o algoritmo retorna um conjunto de novos esquemas XML. O usuário pode escolher qual esquema melhor se adapta às suas necessidades.

ABSTRACT

The use of XML documents for representation, storage, and transport of data it is increasing every day. Many organizations use XML documents to store their information, usually in very large repositories. XML schemas help to organize these documents, imposing restrictions to the way the documents are formed. Updates to the XML documents can be necessary to fulfill new needs. These updates can violate the schema corresponding to the updated XML documents. So, there exists the necessity of to performing updates to these schemas, without invalidating the XML documents recognized previously. Given the need to work in an automated way with those updates in the schemas, this work proposes an algorithm of incremental evolution of schemas for XML. Our algorithm is incremental in the sense of being preservative to the characteristics present in the old schema. The process of incremental evolution in an XML schema D it is carried through from a update list L accomplished in a XML document X that is recognized by the schema D. For each update μ in L, the schema D is evolved, so that it accepts the documents that are in agreement with this update. After the evolution process on each update in L, the algorithm returns a set of new XML schemas. The user can choose the one which better adapts to their needs.

CAPÍTULO 1

INTRODUÇÃO

A utilização da linguagem XML (*eXtensible Markup Language*) para representação e intercâmbio de dados vem aumentando a cada dia. A linguagem XML é vista como formato universal para utilização em documentos estruturados [Chi00]. XML foi projetada para trabalhar com o desafio de publicação e troca de dados em larga escala na Web.

O grande sucesso da linguagem XML é devido às suas características. XML provê uma especificação aberta e independente de arquitetura, sendo ideal para trabalhar em ambientes altamente heterogêneos, os quais possuem diferentes arquiteturas de *hardware* e de $software$.

Devido à grande variedade de dados que podem ser representados, existe a conveniência em se definir esquemas para manipulação e armazenamento de dados em XML. Uma linguagem de esquema para XML descreve restricões para a escrita de documentos XML.

Podemos imaginar a utilização dos documentos XML em dois ambientes. O primeiro seria o armazenamento de informações, onde dados sobre os negócios das organização seriam armazenados e organizados utilizando documentos e esquemas XML. Outro ambiente seria a troca de informações, onde os documentos XML poderiam ser utilizados por diversos sistemas, independentemente de plataforma.

1.1 Motivação

Atualmente, muitas organizações utilizam documentos XML para armazenar suas informações. Geralmente a quantidade de documentos que precisam ser armazenados é grande. Os esquemas XML ajudam a organizar estes documentos, impondo restrições de como eles devem ser formados.

Estes esquemas XML estão diretamente ligados às estruturas que os administradores dos documentos XML querem que estes documentos respeitem. Mudanças nestas estrutu-

ras são inevitáveis e podem surgir por diversos motivos: alteração do universo de discurso, mudanças de interpretação dos fatos, alterações para melhorar o desempenho e correções de erros, entre outros.

No ambiente onde os documentos XML são utilizados para armazenamento, uma possível solução para este cenário de mudanças nas necessidades de armazenamento envolvendo os documentos XML, é alterar o esquema XML e todos os documentos anteriores de maneira a deixar todos os documentos, sejam os antigos como os novos, de acordo com o novo esquema, que satisfaz as novas necessidades.

Estas alterações, em todos os documentos anteriores, podem ser muito complicadas, visto a quantidade de documentos antigos, e também pode haver casos onde estas modificações não sejam possíveis de serem realizadas.

Outra solução é realizar as modificações somente no esquema. Para isto, estas modificações devem ser conservativas, ou seja, o novo esquema deve reconhecer os documentos que estão de acordo com as alterações realizadas, e também manter a compatilidade com os documentos antigos.

Dada a necessidade de trabalhar de forma automatizada com essas alterações nos esquemas, este trabalho propõe um algoritmo de evolução incremental de esquemas para XML. Incremental no sentido de ser conservativa às características presentes no esquema antigo.

A proposta de algoritmo de evolução de esquemas trabalha com a linguagem de definição de esquemas DTD (*Document Type Definition*). Inicialmente o esquema XML é representado na forma de autômato de árvore[1]. Esta transformação de DTD para autômato de árvore é realizada através de um algoritmo apresentado na seção 2.4.

O processo de evolução incremental em um esquema XML D é realizado a partir de modificações L realizadas em um documento XML X, que é reconhecido por D. Para cada modificação μ em L, o esquema D é evoluído para que aceite os documentos que estejam de acordo com estas modificações.

Cada modificação μ contém o nome da operação (inserção ou remoção de atributos ou

[1]Os autômatos de árvores possibilitam uma maneira de representar linguagens representadas por árvores, especialmente, esquemas para XML. Mais informações podem ser encontradas na Seção 2.6.

de elementos XML), a posição[2] pj no documento XML onde foi realizada a modificação, e atributos adicionais de acordo com o tipo da operação.

Após o processo de evolução sobre cada modificação em L, o algoritmo retorna um conjunto de novos esquemas XML. O usuário pode escolher qual esquema melhor se adapta às suas necessidades.

Este processo de evolução dos esquemas XML é realizado pelo algoritmo *ISE*, apresentado na Seção 3.1. Os parâmetros de entrada deste algoritmo são: um autômato de árvore $\mathcal{A}$ correspondente ao esquema original; o documento XML original representado na forma de uma árvore rotulada t, como apresentado na Seção 2.6; e uma lista de modificações L realizadas no documento XML original.

A lista de modificações L pode ter elementos de quatro tipos: inserção de uma sub-árvore t' em uma determinada posição p da árvore original t; inserção de um atributo em uma posição p da árvore original t; exclusão de uma sub-árvore t na posição p da árvore original t; e exclusão de um atributo em uma posição p da árvore original t. Na Seção 3.1 é apresentada, de forma mais detalhada, como esta lista deve ser formada.

O algoritmo *ISE* trabalha de acordo com cada um dos tipos de modificações presentes em L. Quando a modificação é do tipo inserção de uma sub-árvore t, na posição pj de $\mathcal{A}$, existe a necessidade de evoluir a expressão regular presente no autômato de árvore $\mathcal{A}$, correspondente à posição p (o nó pai da posicao pj, como será visto na Seção 2.6). O algoritmo *Evolution-e*, apresentado na Seção 4.3, é usado para este propósito. Os outros casos são facilmente resolvidos, como é apresentado no Capítulo 3.

1.2 Objetivos

O principal objetivo deste trabalho é propor uma abordagem para a evolução incremental de esquemas XML, utilizando expressões regulares e autômatos de árvore. Mais detalhadamente os objetivos são:

- A partir do estudo de propostas sobre autômatos de árvores, transformação de ex-

[2]Cada posição pj representa um nó na árvore XML (documento XML). No Capítulo 2.6 é apresentado, de forma mais detalhada, como esta posição é atribuída aos nós dos documentos XML.

pressões regulares em autômatos finitos e evolução de expressões regulares, modelar um algoritmo que trabalhe com o problema de evolução incremental de esquemas XML.

- Implementar a abordagem modelada, para poder realizar os testes e validar a mesma.
- Realizar um estudo de caso, de maneira a demonstrar na prática a utilização da proposta, bem como poder medir sua qualidade.

1.3 Conteúdo deste documento

O restante deste trabalho está organizado da seguinte maneira: No Capítulo 2 são apresentados os fundamentos utilizados neste trabalho, tais como XML e esquemas para XML, expressões regulares, transformações entre expressões regulares e autômatos finitos, aprendizado de linguaguens regulares e autômatos finitos de árvores.

Nos Capítulos 3 e 4 são apresentados dois algoritmos que são as contribuições deste trabalho. O algoritmo de evolução incremental de esquemas para XML (*ISE*) no Capítulo 3, e o algoritmo de evolução de expressões regulares (`dGREC`) no Capítulo 4.

Um estudo de caso, demostrando a utilização do algoritmo de evolução de esquemas para XML é apresentado no Capítulo 5. No Capítulo 6 são apresentadas as conclusões do trabalho. Ao final deste documento são apresentados os Capítulos 7 e 8, que demonstram testes de desempenho e como foram implementados os algoritmos de evolução incremental de esquemas para XML e evolução de expressões regulares, respectivamente.

CAPÍTULO 2

FUNDAMENTOS

Neste capítulo são apresentados conceitos e definições utilizados neste trabalho. Inicialmente uma visão geral sobre XML e esquemas para XML é apresentada. A seguir são apresentados alguns conceitos e funções sobre expressões regulares. Transformações entre expressões regulares e autômatos finitos são apresentadas nas Seções 2.3 e 2.4. Na Seção 2.5 é apresentada uma visão geral sobre aprendizado de linguagens regulares. Ao final do capítulo, é apresentada uma forma de representação de esquemas para XML, os autômatos de árvores regulares.

2.1 XML e Esquemas para XML

Nesta seção é apresentada uma visão geral sobre XML (*Extensible Markup Language*) e a linguagem DTD (*Document Type Definition*) de esquemas para XML. Um estudo mais detalhado sobre estas linguagens pode ser encontrado em [Mar99], [BPSM$^+$04] e [Har01].

Inicialmente a Web[1] foi criada para a publicação de documentos científicos. Atualmente ela serve de base para inúmeras aplicações como comércio eletrônico, bancos on-line, fóruns, entre outros. Um dos maiores meios de comunicação utilizado na Web é o uso de páginas Web, com as quais o usuário pode visualizar e interagir com as páginas. Atualmente a grande maioria destas páginas é feita usando HTML (*HyperText Markup Language*) [RHJ99]. Segundo [RHJ99] o termo *hypertext* representa textos que têm links para outros textos e o termo *markup language* define anotações para a estrutura de um texto.

Os documentos HTML foram criados para prover estrutura lógica à informação destinada à apresentação de páginas na rede mundial de computadores. A linguagem HTML contém um número fixo de *tags* (ou rótulos) para definir a estrutura de um documento,

[1] O termo Web é usado neste texto como abreviatura de World Wide Web. Este termo é usado para representar a rede mundial de computadores.

e cada *tag* tem a sua semântica já definida. A seguir é apresentado um exemplo de um documento HTML:

```
<html>
<head>
<title>Informações</title>
</head>
<body>
<h1>Informações Pessoais</h1>
<ul>
  <li><b>Nome:</b> João Antônio</li>
  <li><b>Endereço:</b> Rua das Araucárias</li>
  <li><b>E-mail:</b> joao_antonio@email.com</li>
</ul>
</body>
</html>
```

No exemplo apresentado são encontradas as seguintes *tags*: `<html></html>` delimita um documento HTML; `<head></head>` representa o cabeçalho do documento, dentro dela fica a *tag* `<title></title>` onde é informado o título da página; na *tag* `<body></body>` fica todo o corpo do documento HTML. As *tags* `<h1></h1>`, `<ul></ul>`, `<li></li>`, `<b></b>` são *tags* de formatação.

Atualmente existem aproximadamente 100 *tags* presentes na linguagem, de acordo com a definição da *World Wide Web Consortium* (W3C)[2]. Este número tem crescido desde a sua criação em 1992 para satisfazer as necessidades de suas aplicações. HTML é reconhecida por milhares de aplicações incluindo browsers, editores, aplicativos de correio eletrônico, banco de dados, gerenciador de contatos, entre outros.

Apesar do grande sucesso da linguagem HTML, ela possui algumas limitações. Devido as suas extensões ela tem se tornada uma linguagem bastante complexa. Existem várias combinações possíveis de *tags*, e o resultado de uma combinação específica pode ser diferente de um browser para outro.

Apesar de todas estas *tags* incluídas na HTML, mais *tags* são necessárias. Aplicações de comércio eletrônico precisam de *tags* que representem referências a produtos, preços, nome, endereço, etc. Mecanismos de consulta precisam de *tags* para palavras-chave e

[2]A W3C é uma organização que mantém o desenvolvimento de muitos padrões usados na Web, o mais conhecido é o HTML.

descrições. Aplicações que envolvem segurança precisam de *tags* para autenticações. A lista de *tags* necessárias ao HTML não é conhecida a priori. Porém, adicionando novas *tags* em uma linguagem pode não ser uma solução satisfatória.

A linguagem XML (*eXtensible Markup Language*) foi criada, tendo como objetivos superar as limitações da HTML, não com objetivo de novidade, mas sim incorporando muitas das características presentes na HTML.

2.1.1 XML (eXtensible Markup Language)

Extensible Markup Language (XML) é uma linguagem de marcação de dados, desenvolvida pelo W3C como um subconjunto da linguagem *Standard Generalized Markup Language* (SGML), principalmente para superar as limitações da HTML. Ela provê um formato para descrever dados semi-estruturados. Isso facilita declarações mais precisas do conteúdo.

Algumas características sobre a linguagem XML, tais como relatadas em [Mar99] e [Har01] são mostradas a seguir:

- pode ser usada como ferramenta para definir linguagens de marcações; foi feita para responder os conflitos provenientes das alterações necessárias no HTML, para atender à demandas futuras;

- é extensível porque não define *tags*, mas permite ao usuário criar as próprias *tags* necessárias às suas aplicações; com isto não é restritiva;

- é uma linguagem usada para descrever e manipular documentos semi-estruturados. Documentos XML não são limitados a livros e artigos, ou a páginas WEB, e pode incluir objetos em aplicações cliente/servidor. Estes documentos são chamados de "representação de dados semi-estruturados", por alguns autores;

- possui uma representação de árvores, de aridade não restrita. XML não especifica como estas árvores devem ser populadas.

- é um mecanismo flexível que acomoda estrutura de aplicações específicas, tratando

da manipulação de informações, incluindo a visualização das mesmas, e também a manipulação da estrutura destas informações;

- descreve a estrutura e a semântica de um documento, e não a sua formatação.

Devido ao fato de XML fazer a separação entre os dados e a apresentação ao usuário, as aplicações que utilizam XML podem ser construídas e atualizadas de maneira mais flexível do que as que utilizam HTML. A comunicação entre estas aplicações fica transparente devido à linguagem XML ser aberta e independente de plataforma e dispositivo.

A linguagem XML descreve uma classe de objetos de dados chamados documentos XML. Documentos XML são compostos por unidades de armazenamento chamadas entidades. Os dados são compostos de caracteres, sendo alguns deles são caracteres normais, e outros formam marcações. Uma marcarção define como é a estrutura lógica nos documentos e como eles são armazenados. XML provê um mecanismo para impor restrições dentro desta estrutura lógica. Geralmente existe uma grande quantidade de marcações em documentos XML, mas como elas são rotuladas de acordo com a sua descrição, toda a sua estrutura é facilmente compreendida.

A seguir é apresentado um exemplo de um documento XML contendo elementos (*clientes, cliente, nome, endereco, e-mail*) e atributos (*id*):

```
<?xml version="1.0"?>
<clientes>
   <cliente id="1">
      <nome>João Antônio</nome>
      <endereco>Rua das Araucárias</endereco>
      <e-mail>joao_antonio@email.com</e-mail>
   </cliente>
</clientes>
```

Para construir blocos em documentos XML são utilizados os elementos. Cada elemento tem um nome e o seu conteúdo. Uma estrutura simplificada de elemento pode ser visualizada a seguir:

$\langle nome \rangle conteudo \langle / nome \rangle$

No campo *nome* deve ser informado o nome da *tag*, e o campo *conteudo* é o conteúdo do elemento. O conteúdo de um elemento é delimitado por uma marcação conhecida por *início de tag* e *fim de tag*. Este mecanismo de marcação é similar ao HTML.

Os *nomes* dos atributos têm algumas restrições na sua estrutura. Eles devem sempre começar com uma letra ou o caracter '_' (*underscore*). O restante do *nome* pode possuir letras, dígitos, o caractere *underscore*, o ponto '.', ou o traço '-'. Espaços em branco não são permitidos na estrutura do *nome*. Também o *nome* não pode iniciar com a palavra 'xml', a qual é uma palavra reservada da linguagem XML. Os nomes são sensitivos com relação a letras maiúsculas e minúsculas.

É possível escrever informações adicionais nos elementos dos documentos XML na forma de *atributos*. Os atributos têm um nome e um valor. Os seus nomes devem seguir as mesmas regras dos nomes dos elementos. Cada elemento pode ter zero ou mais elementos entre as suas *tags* de abertura e fechamento.

Os elementos que não possuem conteúdo são chamados de elementos vazios. Geralmente, eles são colocados como valores de seus atributos. A estrutura de um elemento vazio é apresentada a seguir:

$\langle nome/\rangle$

O conteúdo de um elemento não é limitado somente a textos. Ele pode conter outros elementos e também textos entre estes elementos. Um documento XML é uma árvore de elementos. Não existe limite para o tamanho da árvore e os elementos podem se repetir.

Seja E um elemento de um determinado documento XML, e seja E' um elemento que está dentro do conteúdo de E. Então dizemos que E' é um elemento filho de E e, E é pai de E'.

Início e fim de *tags* devem ser sempre balanceados, e os elementos filhos devem estar sempre incluídos completamente dentro de seus respectivos pais. Ou seja, não é possível que o final de uma *tag* de um elemento filho apareça depois do final da *tag* de seu respectivo pai. Um documento XML que cumpre com todas essas condições é chamado de *bem formado*.

Todo documento XML deve ter um elemento raiz, e este elemento deve ser único. Em outras palavras, todos os outros elementos do documento devem ser filhos de um mesmo elemento pai.

Para identificar um determinado documento como sendo um documento XML é acrescentado no início do mesmo a *declaração XML*, a qual, além de identificar o documento como um documento XML, também identifica a versão da XML usada neste documento. Esta declaração pode ser visualizada a seguir:

```
<?xml version="1.0"?>
```

A declaração pode conter outros atributos tais como a codificação de caracteres utilizada no documento. Esta declaração XML é opcional. Entretanto, as recomendações XML sugerem que ela seja colocada em todo arquivo XML.

XML permite que sejam inventados nomes de diferentes elementos e atributos de acordo com a necessidade. Estes elementos e atributos devem seguir algumas regras para que o documento seja bem formado. Se um documento não é bem formado, a aplicação que está lendo o mesmo deve retornar "erro".

2.1.2 Esquemas para XML

Uma linguagem de esquema para XML descreve um conjunto de regras de escrita de documentos. Estas linguagens descrevem restricões para a escrita de documentos XML. Estas restrições são chamadas *restrições de esquema*. A *Document Type Definition* (DTD) é a linguagem original para modelagem de esquemas para XML. Entretanto, por razões históricas, a DTD é um pouco limitada, e se tem buscado desenvolver soluções que superem estas limitações.

A DTD como definida em [BPSM$^+$04] é uma gramática de árvore local. Nela não é possível distinguir entre símbolos *terminais* e *não-terminais*. Uma DTD lista os elementos, atributos, entidades, e notações que podem ser usadas em um documento, bem como seus possíveis relacionamentos.

As regras presentes na DTD podem ser usadas para validar os dados quando a aplicação que os recebe não possui internamente uma descrição do dado que está recebendo. Um analisador de documentos pode verificar os dados a partir da análise das regras contidas na DTD para ter certeza de que o dado está estruturalmente correto. Quando um documento não possui uma DTD, ele pode ser usado para implicitamente se auto-descrever, mas neste caso a verificação das regras não pode ser feita.

A sintaxe das DTDs é diferente da sintaxe dos documentos XML. A DTD descreve cada objeto (elementos, atributos) que pode aparecer nos documentos. A leitura de uma declaração de elementos é de fácil entendimento. Em outras palavras, o conteúdo lista os filhos que são aceitos pelo elemento. Uma estrutura simplificada de um elemento de uma DTD pode ser visualizada a seguir:

```
<!ELEMENT loja (livro+)>
```

Depois da marcação `<!ELEMENT` vem o nome do elemento, que segue a mesma estrutura e restrições dos nomes dos elementos dos documentos XML. Depois do nome aparece o conteúdo deste elemento, ou seja descreve os filhos que são aceitos por este elemento. O exemplo apresentado mostra que o elemento *loja* pode possuir um ou mais elementos *livro* como filhos.

O conteúdo de um elemento pode assumir dados simples ou compostos. Estes dados podem ser caracterizados mediante as palavras reservadas e operadores, como descrito a seguir:

- `#PCDATA` representa que o conteúdo de um determinado elemento deve ser texto. Ele representa elementos folhas na árvore de documento XML. Um elemento folha é um elemento que não contém filhos.
- `EMPTY` representa que um determinado elemento é um elemento vazio. Esta declaração sempre indica que um elemento é um elemento folha.
- `ANY` indica que um elemento pode conter qualquer outra declaração de elemento presente na DTD.

Indicadores de ocorrência são usados no conteúdo de um elemento para indicar como seus elementos filhos devem aparecer ou ser repetidos. Estes indicadores podem ser os seguintes caracteres:

- O símbolo '+' indica que um determinado elemento deve aparecer uma ou mais vezes dentro do conteúdo de seu elemento pai.
- O símbolo '*' indica que o elemento pode aparecer zero ou mais vezes.
- O símbolo '?' indica que um determinado elemento é opcional, podendo aparecer uma ou nenhuma vez.

Para separar e indicar a ordem que os filhos devem aparecer no conteúdo de um elemento são utilizados os caracteres de conexão, que são mostrados a seguir:

- O símbolo ',' separa os elementos, e a ordem que eles devem aparecer no documento é a mesma ordem que eles aparecem dentro da especificação na DTD.
- O símbolo '|' indica que somente um dos elementos, ou o da direita ou da esquerda do caractere '|' deve aparecer.

Os atributos de um determinado elemento também devem ser declarados na DTD. Eles são declarados com a declaração `ATTLIST`. Um exemplo de declaração de um atributo pode ser visualizado a seguir:

```
<!ATTLIST pedido situacao (aberto|entregue)  "aberto" >
```

A declaração deste atributo é formada pela marcação `<!ATTLIST`, o nome do elemento (pedido), o nome do atributo (situacao), o tipo do atributo ((aberto|entregue)), e o valor padrão ("aberto"), e termina com o caractere >.

Os atributos são divididos em três categorias:

- os atributos que contêm somente texto, onde é usada a palavra `CDATA` para representá-los, por exemplo:

  ```
  <!ATTLIST cliente nome CDATA  #REQUIRED >
  ```

A palavra `REQUIRED` representa que este atributo é obrigatório.

- atributos com restrições no seu conteúdo, como exemplo:

```
<!ATTLIST cliente id ID  #IMPLIED >
```

A palavra reservada `ID` representa um identificador, ou seja, o valor presente neste atributo deve ser único em todo documento. E a palavra `#IMPLIED` indica que se nenhum valor for colocado neste atributo, a aplicação que está lendo o documento pode definir um valor por ela mesma.

- e os atributos que possuem valores pré-definidos, como exemplo:

```
<!ATTLIST pedido situacao (aberto|entregue)  "aberto" >
```

Além de `CDATA` e `ID`, o tipo de um determinado atributo pode assumir os seguintes valores:

- `ENTITY` é um nome de uma entidade externa, que é usado para ligar este atributo com uma entidade externa.
- `ENTITIES` representa uma lista de `ENTITY` separada por espaços.
- `IDREF` deve ser um valor de um atributo identificador (`ID`) usado em outro lugar no mesmo documento. Este tipo de atributo é usado para ligar atributos no mesmo documento.
- `IDREFS` é uma lista de `IDREF` separada por espaços.
- `NMTOKEN` representa que o valor de um atributo deve ter apenas uma palavra, sem espaços.
- `NMTOKENS` é uma lista de `NMTOKEN` separada por espaços.

A palavra `#FIXED` é usada para fixar um determinado valor padrão a um atributo. Não é necessário que este valor seja informado no documento XML, neste caso o programa leitor

do documento assumirá o valor informado na DTD. Caso o atributo seja informado no documento XML, o valor deve ser igual ao especificado na DTD. Se for um valor diferente, o programa leitor retornará "erro". A palavra `#FIXED` pode ser usada por exemplo para fixar o formato de tamanho de um arquivo, como no exemplo a seguir:

```
<!ATTLIST arquivo_tamanho formato NMTOKEN #FIXED "Megabytes" >
```

Para os elementos que têm mais de um atributo, seus atributos podem ser agrupados. O seguinte exemplo de um elemento cliente, possui dois atributos, o *id* que é um identificador do cliente, e o *nome* que representa o nome do cliente:

```
<!ATTLIST cliente id ID  #IMPLIED nome CDATA #REQUIRED >
```

No exemplo *clientes.dtd*, onde é apresentada uma DTD para dados de um cliente, é possível visualizar a estrutura de uma DTD, contendo regras de elementos e de atributos. No exemplo *clientes.xml* é apresentado um documento XML que satisfaz a DTD *clientes.dtd*. Podemos dizer que *clientes.xml* é gerado a partir da gramática da DTD *clientes.dtd*.

Exemplo 1 (clientes.dtd)

```
<!ELEMENT clientes (cliente*)>
<!ELEMENT cliente (nome,endereco,e-mail)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT endereco (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
<!ATTLIST cliente id ID #REQUIRED>
```

Exemplo 2 (clientes.xml)

```
<?xml version="1.0"?>
<!DOCTYPE clientes SYSTEM "clientes.dtd">
<clientes>
   <cliente id="1">
      <nome>João Antônio</nome>
      <endereco>Rua das Araucárias</endereco>
      <e-mail>joao_antonio@email.com</e-mail>
   </cliente>
</clientes>
```

Em *clientes.xml* é possível visualizar a declaração XML de tipo (*XML document type declaration*), que é usada para representar que um determinado documento deve satisfazer a DTD especificada. Esta declaração pode ser visualizada a seguir:

```
<!DOCTYPE clientes SYSTEM "clientes.dtd">
```

Também existe a possibilidade da DTD estar contida dentro do próprio documento XML como apresentado no exemplo *clientes-dtd.xml*.

Exemplo 3 (clientes-dtd.xml)

```
<?xml version="1.0"?>
<!DOCTYPE clientes [
<!ELEMENT clientes (cliente*)>
<!ELEMENT cliente (nome,endereco,e-mail)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT endereco (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
<!ATTLIST cliente id ID #REQUIRED>
]>
<clientes>
   <cliente id="1">
      <nome>João Antônio</nome>
      <endereco>Rua das Araucárias</endereco>
      <e-mail>joao_antonio@email.com</e-mail>
   </cliente>
</clientes>
```

Este trabalho vai se concentrar apenas na linguagem de esquemas DTD, por ser de amplo uso e fácil de ser verificada. Porém, a técnica apresentada neste trabalho para evolução de esquemas para XML, eventualmente, poderá ser estendida para outras linguagens de esquema, que possam ser reconhecidas usando autômatos de árvore.

2.2 Expressões Regulares

Nesta seção é apresentada uma introdução a expressões regulares, bem como, algumas funções úteis para manipulá-las. Uma abordagem mais detalhada pode ser encontrada na referência [CD00].

Seja Σ um conjunto finito de símbolos chamado *alfabeto*. Os elementos de Σ são chamados de *letras* representados por $a, b, c, ...$ Uma palavra sobre o alfabeto Σ é uma sequência finita de símbolos de Σ que pode ser escrita como $a_1a_2...a_n$. O símbolo ε representa a palavra vazia. O tamanho de uma palavra w é representado por $|w|$. O conjunto de todas as palavras sobre o altafeto Σ é representado por Σ^*. Σ^+ é gerado a partir de $\Sigma^* - \{\varepsilon\}$. A linguagem vazia é representada por $\emptyset$.

Sejam L e L' duas linguagens sobre Σ. Abaixo são apresentadas várias operações possíveis entre estas linguagens:

- $L \cup L'$ representa a união das linguagens L e L';
- LL' representa a concatenação das linguagens L e L';
- $L^0 = \{\varepsilon\}$;
- $L^{n+1} = L^nL$, para $n \geq 0$, onde n representa a n-ésima potência da linguagem L;
- L^* representa o fecho de Kleene da linguagem L;
- $L?$ representa linguagem L;
- $L \cap L'$ representa a intersecção das linguagens L e L'.

Segundo [Stu03] uma expressão regular é uma palavra contendo uma combinação de caracteres normais e *meta-caracteres*, na qual os *meta-caracteres* são caracteres que representam idéias de quantidade, localização ou tipo de caracteres.

Uma expressão regular E sobre o alfabeto Σ é definida recursivamente da seguinte maneira:

- $\emptyset$ é a expressão regular que representa a linguagem vazia;
- ε é a expressão regular que representa a linguagem $\{\varepsilon\}$;
- a é a expressão regular que representa a linguagem $\{a\}$, onde $a \in \Sigma$;
- se E e F são expressões regulares que representam respectivamente as linguagens $L(E)$ e $L(F)$, então as seguintes expressões também são expressões regulares:

- $(E) + (F)$ representa a linguagem $L(E) \cup L(F)$;
- $(E), (F)$ representa a linguagem $L(E)L(F)$;
- $(E)^*$ representa a linguagem $\bigcup_{i=0}^{\infty} L^i(E)$;
- $(E)^+$ representa a linguagem $\bigcup_{i=1}^{\infty} L^i(E)$;
- $(E)?$ representa a linguagem $\bigcup_{i=0}^{1} L^i(E)$.

Segundo [HU79] o formato de uma expressão regular E sobre o alfabeto Σ pode ser definido como: $E ::= \emptyset \mid \varepsilon \mid \alpha_i \mid E + E \mid E.E \mid E^+ \mid E^* \mid E? \mid (E)$. Onde ε representa a palavra vazia, e α_i representa uma palavra qualquer α com um índice i. Dada a expressão regular E, para determinar a ordem que os símbolos aparecem em E, eles são indexados seguindo a sua ordem de leitura. Por exemplo, dada uma expressão regular $E = (a|b)^*c^+d$, a expressão indexada é representada por $\bar{E} = (a_1|b_2)^*c_3^+d_4$.

O conjunto de posições da expressão regular E é representado por $Pos(E)$. A função *symb* retorna o símbolo que está na posição $Pos(E)$ de E. O alfabeto indexado de $\bar{E}$ é representado por $\alpha = \{\alpha_1, ..., \alpha_n\}$, onde $n = |Pos(E)|$ (número de posições em E). Para simplificar a representação de uma expressão regular indexada $\bar{E}$, iremos utilizar simplesmente E.

A seguir são apresentadas algumas funções sobre expressões regulares introduzidas em [CD00].

Definição 2.2.1 *$Null_E$ pode ser indutivamente computada, a qual assume $\{\varepsilon\}$ se ε pertence a L(E) e $\emptyset$ caso contrário.*

$$
\begin{array}{lcl}
Null_{\emptyset} & = & \emptyset \\
Null_{\varepsilon} & = & \{\varepsilon\} \\
Null_{a} & = & \emptyset \\
Null_{F+G} & = & Null_F \cup Null_G \\
Null_{F.G} & = & Null_F \cap Null_G \\
Null_{F^+} & = & Null_F \\
Null_{F^*} & = & \{\varepsilon\} \\
Null_{F?} & = & \{\varepsilon\}
\end{array}
$$

Null_E pode ser computada em tempo linear, do tamanho de E.

Definição 2.2.2 *First*(E) é o conjunto das posições iniciais das palavras da linguagem $L(E)$.

$First(E) = \{x \in Pos(E) \mid \exists u \in \Sigma^* : \alpha u \in L(E) \wedge symb(x) = \alpha\}$

First(E) pode ser recursivamente computada da seguinte maneira*:*

$$\begin{array}{lcl} First(\emptyset) & = & \emptyset \\ First(\varepsilon) & = & \emptyset \\ First(\alpha_x) & = & x \\ First(F+G) & = & First(F) \cup First(G) \\ First(F.G) & = & First(F) \cup Null_F.First(G) \\ First(F^+) & = & First(F) \\ First(F^*) & = & First(F) \\ First(F?) & = & First(F) \end{array}$$

A computação da operação de união que aparece na equação anterior pode ser executada em tempo constante. Isto porque estes conjuntos são disjuntos, desde que eles pertençam a diferentes partes da expressão regular. Portanto, $First(E)$ pode ser executada em tempo linear, no tamanho de E.

Definição 2.2.3 *Last*(E) é o conjunto das posições finais das palavras da linguagem $L(E)$.

$Last(E) = \{x \in Pos(E) | \exists u \in \Sigma^* : u\alpha \in L(E) \wedge symb(x) = \alpha\}$

Last(E) pode ser recursivamente computada da seguinte maneira:

$$\begin{array}{lcl} Last(\emptyset) & = & \emptyset \\ Last(\varepsilon) & = & \emptyset \\ Last(\alpha_x) & = & x \\ Last(F+G) & = & Last(F) \cup Last(G) \\ Last(F.G) & = & Last(G) \cup Null_G.Last(F) \\ Last(F^+) & = & Last(F) \\ Last(F^*) & = & Last(F) \\ Last(F?) & = & Last(F) \end{array}$$

Como na definição anterior, $Last(E)$ pode ser computada em tempo linear, do tamanho de E.

Para cada conjunto X, a função $\mathcal{I}_\mathcal{X}$ é definida de X para {{ε},$\emptyset$} como:

$$\mathcal{I}_\mathcal{X}(x) = \left\{ \begin{array}{ccc} \emptyset & if & x \notin X \\ \{\varepsilon\} & if & x \in X \end{array} \right.$$

Esta função é utilizada nas definições seguintes, para selecionar posições da expressão regular.

Definição 2.2.4 *Follow*(E,x) é o conjunto das posições que estão imediatamente após a posição x na expressão E. Se x não pertence às posições de E então *Follow*(E,x) $= \emptyset$.

$Follow(E,x) = \{y \in Pos(E) | \exists v, w \in \Sigma^* : v\alpha_1\alpha_2 w \in L(E) \wedge symb(x) = \alpha_1 \wedge symb(y) = \alpha_2\}$

Follow(E,x) pode ser recursivamente computada da seguinte maneira:

$$
\begin{array}{lcl}
Follow(\emptyset, x) & = & \emptyset \\
Follow(\varepsilon, x) & = & \emptyset \\
Follow(a, x) & = & \emptyset \\
Follow(F + G, x) & = & \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\
Follow(F.G, x) & = & \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \cup \\
 & & \mathcal{I}_{Last(F)}(x).First(G) \\
Follow(F^+, x) & = & Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F) \\
Follow(F^*, x) & = & Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F) \\
Follow(F?, x) & = & Follow(F, x)
\end{array}
$$

Definição 2.2.5 $Previous(E, x)$ é o conjunto das posições que precedem a posição x na expressão E. Se $x \notin Pos(E)$ então $Previous(E, x) = \emptyset$.

$Previous(E, x) = \{y \in Pos(E) | \exists v, w \in \Sigma^* : v\alpha_2\alpha_1 w \in L(E) \wedge symb(x) = \alpha_1 \wedge symb(y) = \alpha_2\}$

A execução de $Previous(E, x)$ é similar a de $Follow(E, x)$ e pode ser recursivamente computada como segue:

$$
\begin{array}{lcl}
Previous(\emptyset, x) & = & \emptyset \\
Previous(\varepsilon, x) & = & \emptyset \\
Previous(a, x) & = & \emptyset \\
Previous(F + G, x) & = & \mathcal{I}_{Pos(F)}(x).Previous(F, x) \cup \mathcal{I}_{Pos(G)}(x).Previous(G, x) \\
Previous(F.G, x) & = & \mathcal{I}_{Pos(F)}(x).Previous(F, x) \cup \mathcal{I}_{Pos(G)}(x).Previous(G, x) \cup \\
 & & \mathcal{I}_{First(F)}(x).Last(G) \\
Previous(F^+, x) & = & Previous(F, x) \cup \mathcal{I}_{First(F)}(x).Last(F) \\
Previous(F^*, x) & = & Previous(F, x) \cup \mathcal{I}_{First(F)}(x).Last(F) \\
Previous(F?, x) & = & Previous(F, x)
\end{array}
$$

As funções $Follow(E, x)$ e $Previous(E, x)$ podem ser implementadas em tempo constante ou linear, usando operações de conjuntos.

2.3 Transformação de Expressão Regular em Autômato Finito de Glushkov

O Algoritmo de Glushkov, [CD00], é utilizado para computar o Autômato de Glushkov a partir de uma expressão regular. Segundo o teorema de Kleene, dada uma expressão regular, existe um autômato finito que reconhece a linguagem que a expressão define. Os Autômatos de Glushkov podem ser utilizados para reconhecer a linguagem de uma determinada expressão regular. Nesta seção é apresentado o algoritmo para a transformação de uma determinada expressão regular em um Autômato de Glushkov.

Um autômato finito não determinístico (NFA) é uma quíntupla $M = (Q, \Sigma, I, F, \delta)$ onde Q é um conjunto finito chamado de conjunto de estados, Σ é um alfabeto, $I \in Q$ é o conjunto de estados iniciais, $F \in Q$ é o conjunto estados finais e $\delta \in Q$ x Σ x Q é o conjunto de transições. O autômato é representado por um grafo rotulado, onde os vértices são os estados dos autômatos e as arestas são as transições(setas) rotuladas. Um determinado autômato é determinístico (DFA) se e somente se existe somente um estado inicial e para todos $(q, a) \in Q$ x Σ existe mais um estado p tal que $(q, a, p) \in \delta$.

Um caminho de tamanho n em M é um sequência $c = f_1 f_2 ... f_n$ de n setas consecutivas $f_i = (p_i, a_i, q_{i+1})$ com i = 1,...,n, e $p_i = q_i$ para i = 2,...,n. O rótulo de um caminho c é a palavra $a_1 a_2 ... a_n$. Uma palavra $w = a_1 a_2 ... a_n$ é reconhecida pelo autômato M se existe um caminho em M rotulado w tal que $p_1 \in I$ e $q_{n+1} \in F$. A linguagem $L(M)$ reconhecida pelo autômato M é o conjunto de todas as palavras reconhecidas pelo próprio M.

O autômato $\bar{M}$ = (Q,σ,{s_I},F,$\bar{\delta}$) que reconhece a linguagem $L(E)$, é construído da seguinte maneira:

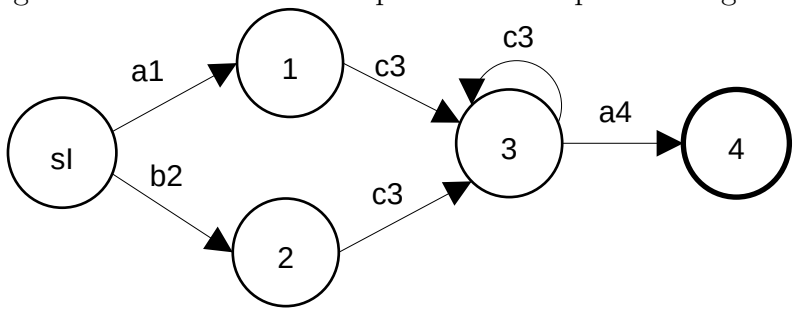
1. $Q = Pos(E) \cup \{s_I\}$
2. $\forall x \in First(E)$, $\bar{\delta}(s_I,\alpha_x) = \{x\}$
3. $\forall x \in Pos(E)$, $\forall y \in Follow(E, x)$, $\bar{\delta}(x,\alpha_y) = \{y\}$
4. $F = Last(E) \cup Null_E.\{s_I\}$

Exemplo 4 (Transformação de Expressão Regular em Autômato de Glushkov) *Como exemplo do processo de transformação de expressão regular em autômato de Glushokv, dada a expressão regular* $E = (a_1{+}b_2), c_3^+, a_4$. *Construíndo o autômato* $M = (Q, \sigma, \{s_I\}, F, \bar{\delta})$ *que reconhece a linguagem* $L(E)$ *temos:*

- $Q = \{s_I, 1, 2, 3, 4\}$, *o conjunto das posições de* E *(*$Pos(E)$*), mais o elemento* s_I, *correspondente ao estado inicial do autômato.*
- $\sigma = \{a_1, b_2, c_3, a_4\}$, *alfabeto de* E.
- $s_I = \{s_I\}$, *conjunto de estados iniciais.*
- $F = \{4\}$, *conjunto de estados finais de* E. *(*$F = Last(E) \cup Null_E.\{s_I\}$*).*
Conjunto de Transições δ		
s_I	a_1	1
s_I	b_2	2
1	c_3	3
2	c_3	3
3	c_3	3
3	a_4	4

A partir dos dados acima, podemos montar uma representação visual do autômato M, *correspondente à expressão regular* E, *como ilustrado na Figura 2.1.*

Figura 2.1: Autômato correspondente à Expressão Regular E



2.4 Transformação de Autômato de Glushkov em Expressão Regular

Neste tópico é apresentada uma solução para a transformação de autômato finito de Glushkov em expressão regular, segundo P. Caron e D. Ziadi [CD00]. Uma explicação mais detalhada sobre o método, bem como, provas sobre o mesmo pode ser encontrada em [CD00].

O método consiste em, a partir de um autômato de Glushkov M, encontrar a expressão regular que se equivale ao autômato M. Para isto é utilizado um grafo G correspondente a M que é definido a seguir. A partir do grafo G são aplicadas regras de redução no mesmo, de maneira a chegar em apenas um estado. Este estado possuirá um rótulo, que será a expressão regular equivalente ao autômato M.

Definição 2.4.1 (Grafo) *Dado um autômato finito homogêneo M, temos o grafo de Glushkov $G = (X,U)$ correspondente a M onde X é o conjunto de vértices (semelhante ao conjunto de estados do autômato) e U é o conjunto de arestas (correspondente a relação de transição δ). O grafo G é direcionado e sem rótulos nas arestas.*

O grafo G tem um nó *root* (nó raiz) r e seu respectivo *antiroot* se existe um caminho de r para qualquer nó do grafo. Um grafo é chamado de *hammock* se ele possui ambos os nós $root(r)$ e $antiroot(s)$, com $r \neq s$.

Definição 2.4.2 (Órbita) *Seja G um grafo. Um conjunto qualquer $\mathcal{O} \subseteq X$ é uma* órbita*, se e somente se, para todos os x e x' pertencentes a $\mathcal{O}$, existe um caminho não trivial de x para x'.*

Propriedades das Órbitas:

- Toda órbita $\mathcal{O}$ contém entradas $In(\mathcal{O})$ e saídas $Out(\mathcal{O})$ que são definidas da seguinte maneira:
  - $In(\mathcal{O}) = \{x \in \mathcal{O} \mid \exists x' \in (X \backslash \mathcal{O}), (x\ ,\ x') \in U\}$
  - $Out(\mathcal{O}) = \{x \in \mathcal{O} \mid \exists x' \in (X \backslash \mathcal{O}), (x'\ ,\ x) \in U\}$
- Uma *órbita* é *maximal* se para cada vértice x pertencente a $\mathcal{O}$ e para cada vértice x' pertencente à saída de $\mathcal{O}$, não existe ao mesmo tempo um caminho de x para x' e um caminho de x' para x. Em outras palavras, uma *órbita* é *maximal* se ela não está contida dentro de outra *órbita*.
- Uma órbita $\mathcal{O}$ é estável sse $\forall x \in Out(\mathcal{O})$ e $\forall y \in In(\mathcal{O})$, existe a aresta (x,y), ou seja, se todas as saídas da órbita estão conectadas às suas entradas.

- Uma órbita $\mathcal{O}$ é *fortemente estável* se $\mathcal{O}$ é estável e após a exclusão das arestas (x,y) $\in Out(\mathcal{O})$ x $In(\mathcal{O})$, cada subórbita maximal é fortemente estável.

- Uma órbita $\mathcal{O}$ é dita transversa, se e somente se:

  - $\forall x, y \in Out(\mathcal{O})$, $\forall z \in (X \backslash \mathcal{O})$ $(x, z) \in U \Rightarrow (y, z) \in U$.
  - $\forall x, y \in In(\mathcal{O})$, $\forall z \in (X \backslash \mathcal{O})$ $(z, x) \in U \Rightarrow (z, y) \in U$.

- Uma órbita $\mathcal{O}$ é fortemente transversa se ela é transversa e se para cada subórbita maximal obtida pela exclusão das arestas (x, y), com $x \in Out(\mathcal{O})$, $y \in In(\mathcal{O})$ também é fortemente transversa.

Um grafo $G = (X, U)$ é dito ser sem órbita, se e somente se $\forall x, y \in X$, não existe simultaneamente um caminho não trivial de x para y e de y para x.

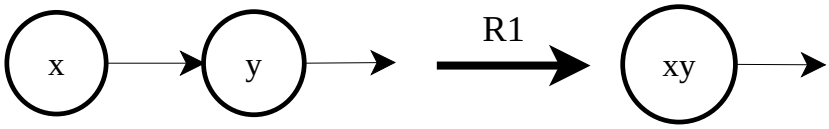
Seja o grafo G no qual todas as órbitas são fortemente estáveis. O grafo $SO(G)$, grafo sem órbita do grafo G, é obtido pela exclusão recursiva de todas as órbitas maximais $\mathcal{O}$ de G, ou seja, exclusão de todas as arestas (x, y), tal que $x \in Out(\mathcal{O})$ e $y \in In(\mathcal{O})$.

Seja x um vértice em $G = (X,U)$. Nós denotamos $Q_G^-(x) = \{y \in X \mid (y, x) \in U\}$ o conjunto dos imediatamente predecessores de x em G e $Q_G^+(x) = \{y \in X \mid (x, y) \in U\}$ o conjunto dos imediatamente sucessores de x em G. Para simplificar será usado $Q^-(x)$ para $Q_G^-(x)$, e $Q^+(x)$ para $Q_G^+(x)$.

Seja G um agrafo sem órbitas. G é dito reduzível se é possível reduzi-lo sucessivamente através das aplicações das regras R_1, R_2, R_3 descritas a seguir.

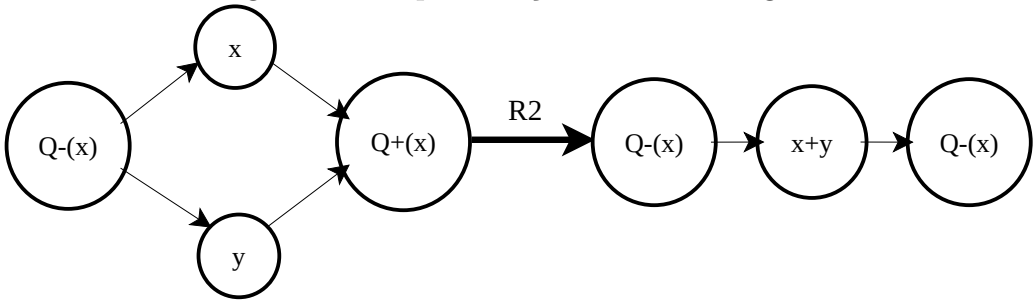
Regra $\mathbf{R_1}$: Sejam dois vértices x e y. Se $Q^-(y) = \{x\}$ e $Q^+(x) = \{y\}$, então é possível "apagar" o vértice y. O vértice x passa a ter o rótulo com a concatenação de x e y (xy). E os sucessores de x ($Q^+(x)$) passam a ser os mesmos de y ($Q^+(y)$).

Figura 2.2: Representação Visual da Regra 1



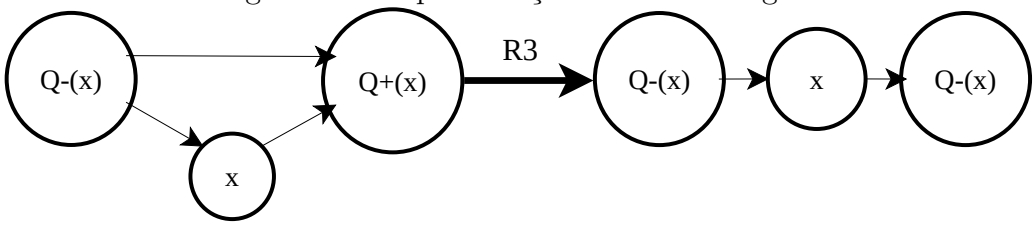
Regra $\mathbf{R_2}$: Sejam dois vértices x e y se $Q^-(x) = Q^-(y)$ e $Q^+(x) = Q^+(y)$, então é possível "apagar" o vértice y, com o vértice x passando a ter o rótulo com a união de x e y (x+y). E com $Q^-(x) = Q^-(x) \cup Q^-(y)$ e $Q^+(x) = Q^+(x) \cup Q^+(y)$.

Figura 2.3: Representação Visual da Regra 2



Regra $\mathbf{R_3}$: Se um vértice x satisfaz as condições de $y \in Q^-(x) \Rightarrow Q^+(x) \subset Q^+(y)$, então é possível apagar as arestas que vão de $Q^-(x)$ para $Q^+(x)$. Com o rótulo de x ficando como $\bar{x}$. Este processo caracteriza um estado como sendo opcional.

Figura 2.4: Representação Visual da Regra 3



Seja $G = (X, U)$ um grafo. G é um grafo de Glushkov se, e somente se, G é um *hammock*; todas as órbitas maximais de G são fortemente estáveis, e fortemente transverssas; e o grafo sem órbita de G é *reduzível*.

A execução do algoritmo de transformação de um autômato $\mathcal{A}$ de Glushkov em uma expressão regular E é feita de acordo com os seguintes passos:

1. Encontrar o Grafo de Glushkov $G = (X,U)$ correspondente ao autômato $\mathcal{A}$.

2. Aplicar as regras R_1, R_2 e R_3 recursivamente sobre G, até o grafo G possuir somente uma aresta. Esta aplicação das regras deve seguir a seguinte restrição:

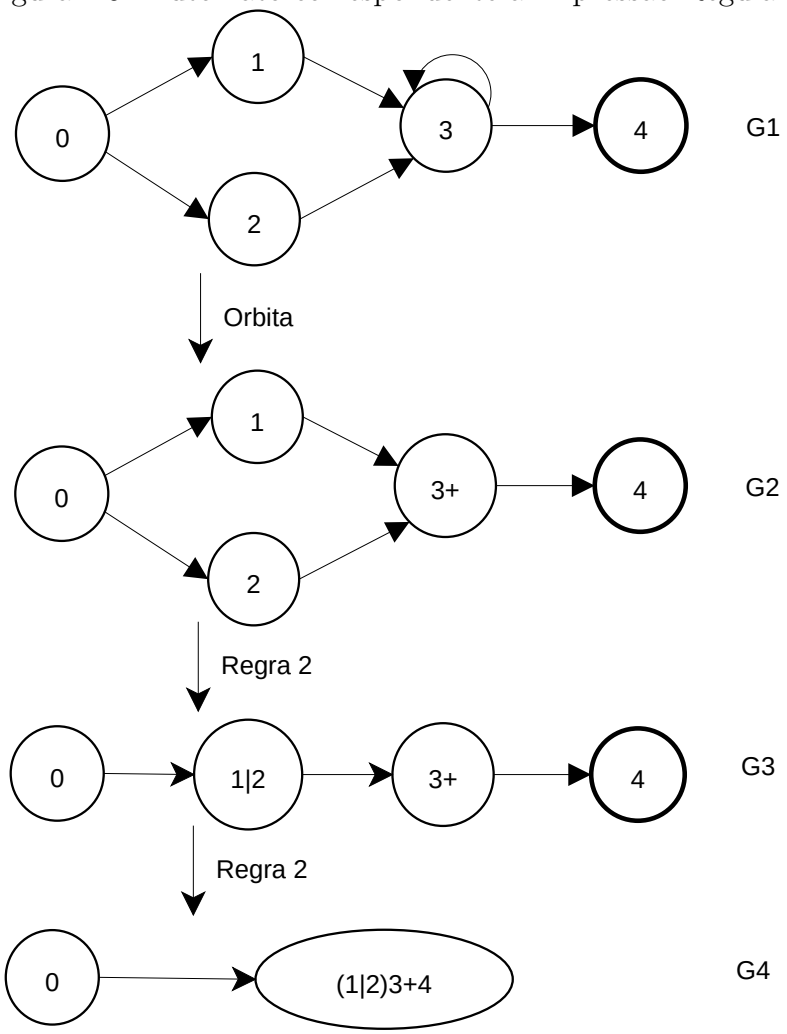
   - Seja $\mathcal{O}$ uma órbita maximal de um grafo G. Pela iteração das regras R_1, R_2, R_3 em $SO(G)$, a órbita $\mathcal{O}$ será reduzida a um único vértice, sobre a restrição

de que as regras R_1 e R_2 são aplicáveis somente em pares $(x, y) \in \mathcal{O}^2$ ou $(x, y) \in (X \setminus \mathcal{O})^2$.

3. Após a realização dos itens acima, o rótulo do único vértice presente em G será a expressão regular que reconhece a gramática reconhecida por $\mathcal{A}$.

Exemplo 5 (Transformação de Autômato de Glushkov em Expressão Regular) *Considerando o autômato gerado pelo Exemplo 4, representado pela Figura 2.1, Seja* $G1$ *o grafo correspondente a este autômato, apresentado na Figura 2.5.*

Figura 2.5: Autômato correspondente à Expressão Regular E



Iniciando o processo da tranformação, primeiramente deve ser trabalhado nas órbitas máximais do grafo $G1$*. Exite uma órbita maximal em* G*, a órbita* $\mathcal{O} = \{\ni\}$*. Como esta órbita possui somente um vértice (*$\{3\}$*), não é necessário realizar o processo de redução*

dentro dela. A transição que vai de 3 *para* 3 *é removida, e o símbolo* + *é adicionado no rótulo do vértice* 3*, resultando no grafo* $G2$ *(Figura 2.5).*

Prosseguindo com a execução do algoritmo no grafo $G2$*, é possível aplicar a Regra 2 nos vértices 1 e 2, resultando no Grafo* $G3$ *(Figura 2.5). Agora, aplicando a Regra 1 duas vezes no grafo* $G3$*, temos a geração do Grafo* $G4$ *(Figura 2.5). Onde existe o vértice com o rótulo* $(1|2)3^{+}4$*, que é a expressão regular* E *gerada. Substituíndo as posições de* E *pelos seus símbolos correspondentes, temos* $E = (a|b)c^{+}a$*.*

2.5 Aprendizado de Linguagens Regulares

Nesta seção é apresentada uma visão geral sobre aprendizado de linguagens regulares, mais especificamente aprendizado em expressões regulares.

Seja $w \in L$ uma palavra que é reconhecida pela expressão E, e seja w' uma palavra igual a w com uma única alteração (inserção ou exclusão de um símbolo na posição p de w). Devido a esta alteração, a nova palavra w' não necessariamente pertence a linguagem L. O objetivo é, a partir de um autômato M_E que reconhece w, encontrar um autômato M'_E que reconheça w', e que continue reconhecendo w.

Considerando que M_E está associado a E, e assumindo que $w' \notin L$, é possível dizer que M_E falha no reconhecimento de w'. A partir desta falha é disparada a criação de uma nova expressão regular E' que estende L para a nova linguagem L', respeitando a semântica de E. Como esta alteração deve ser conservativa, tem-se que $L \subseteq L'$. Diferentes mudanças na nova expressão regular são propostas para o usuário, para que este seja capaz de decidir qual a que melhor se encaixa em sua aplicação.

Seja E uma expressão regular, seja w uma palavra em $L(E)$, e seja w' a palavra obtida depois da inserção ou exclusão de um símbolo na posição p de w. A nova expressão regular a ser obtida satisfaz as condições de $L(E) \cup \{w'\} \subseteq L(E')$ e $\mathcal{D}(E, E') = 1$, ou seja a alteração em E' com relação a E é mínima, com inserção ou exclusão de um símbolo. $\mathcal{D}$ representa a diferença na quantidade de símbolos entre duas expressões regulares.

São consideradas duas possíveis alterações nas palavras. A operação $Del(w, p) = \alpha\beta$, onde $w = \alpha\sigma\beta$ e $|\alpha| = p$, retorna a palavra obtida pela remoção do símbolo σ na posição p

da palavra w. A operação $Ins(w, \sigma, p) = \alpha\sigma\beta$, onde $w = \alpha\beta$ e $|\alpha| = p$, insere um símbolo σ na posição p da palavra w.

O algoritmo `GREC` (*Generate Regular Expression Choices*) [BDA+04] gera, de maneira conservativa, novas expressões regulares E' que estendem uma linguagem L representada pela expressão regular E. Ele é uma extenção do processo de redução de um grafo de Glushkov proposto em [CD00], método abordado na Seção 2.4, para transformar um autômato de Glushkov em uma expressão regular. Mais informações, provas, e uma abordagem mais detalhada pode ser encontrada em [BDA+04].

Dada a expressão regular E, a palavra $w \in L(E)$, e w' uma palavra igual a w com uma alteração de inserção ou exclusão, apresentamos a seguir como o algoritmo `GREC` trabalha com cada caso.

Para o caso da exclusão, o processo é simples, e é realizado da seguinte maneira: Dada uma expressão regular E, o algoritmo de Glushkov [CD00] é utilizado para obter o autômato de Glushkov M_E que aceita $L(E)$. M_E tem um estado para cada posição correspondente na expressão E. Seja s o estado de M_E que corresponde ao símbolo que foi excluído de w. Para obter o novo autômato M'_E, que aceita $L(E')$, M_E é modificado adicionando novas transições de todos os predecessores de s para todos os sucessores de s, ou seja, tornando o estado s como opcional. A nova expressão regular E' é então obtida pelo processo de redução do autômato M'_E, usando o processo apresentado na Seção 2.4.

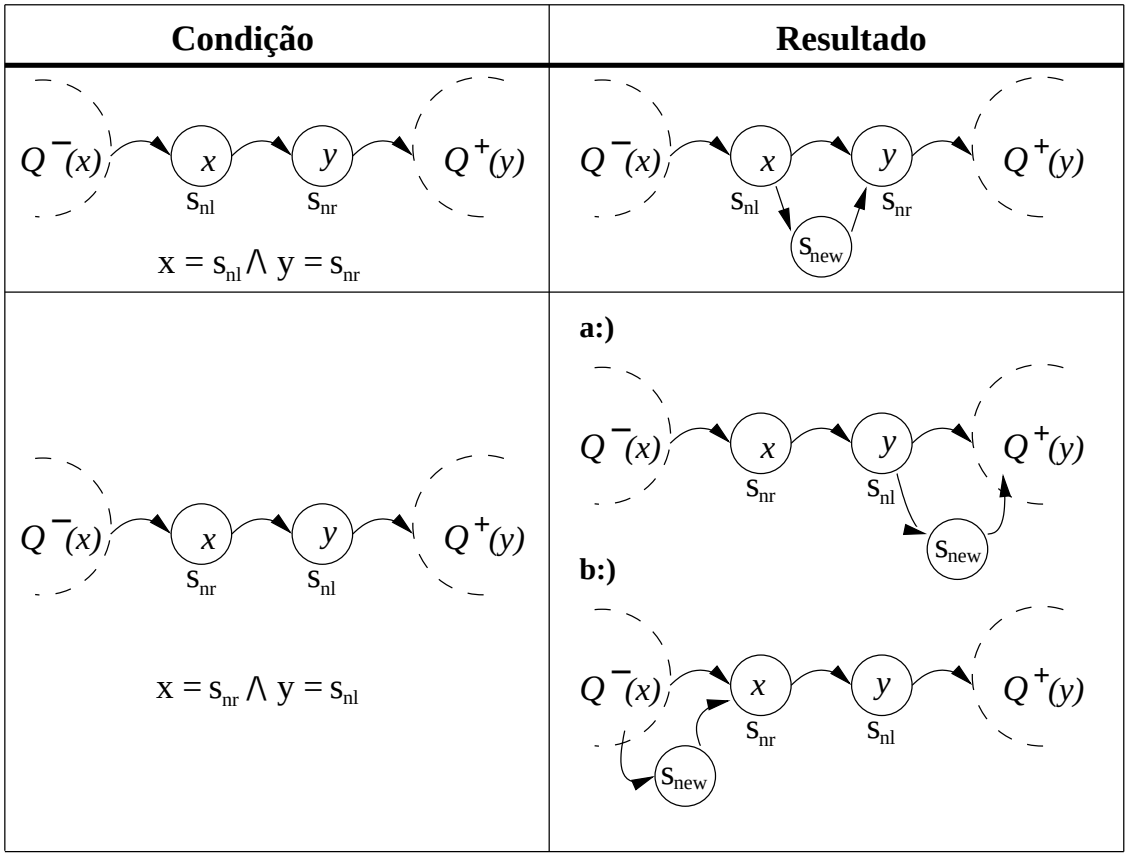
Como exemplo do processo de exclusão, dada uma expressão regular $E = ab^+c$, e a palavra $w = abc$, excluindo o símbolo b temos $w' = ac$, a nova expressão regular E' é construída tornando o símbolo b como opcional na expressão E, assim temos $E' = ab^*c$.

No caso da inserção, o processo é mais detalhado. Considere a execução do autômato M_E sobre a palavra w' e seja p a posição de w' onde o novo símbolo foi inserido. Seja *nearest left state* (s_{nl}) o estado em M_E que é alcançado depois da leitura do primeiro símbolo $p-1$ em w' ou w. Seja *nearest right state* (s_{nr}) o estado em M_E que é sucessor de s_{nl} quando é ignorado o novo símbolo da palavra.

O algoritmo `GREC` considera somente a inserção de uma nova posição em E, que claramente corresponde à inserção de um novo estado s_{new} em M_E. Esta inserção é realizada

de acordo com as situações de s_{nl} e s_{nr} em M_E e as condições sobre cada regra do processo de redução do grafo. A seguir serão apresentadas as situações de s_{nl} e s_{nr} sobre as regras R_1, R_2 e R_3, e as devidas modificações para a inserção do novo estado s_{new}.

Figura 2.6: Condições e modificações para quando for aplicado a Regra R_1



Para as condições e modificações quando for aplicada a regra R_1, existem dois casos que podem ser visualizados na Figura 2.6, esses casos são detalhados a seguir:

1. no primeiro caso tem-se $x = s_{nl}$ $y = s_{nr}$. Neste caso o estado s_{new} é inserido como um nó opcional entre s_{nl} e s_{nr}. Por exemplo, dada uma expressão regular $E = ab$, e a palavra w que é aceita por E, e seja $w' = anb$ a palavra w com uma modificação. As expressões regulares E' geradas neste caso seriam an^*b e $an?b$.

2. no segundo caso s_{nl} é saída de uma órbita e s_{nr} pertence à entrada da mesma órbita (isto somente nas situações nas quais as condições de transformações são satisfeitas). Duas diferentes modificações podem ser propostas neste caso. O estado s_{new} herda as características do estado onde ele foi inserido, ou seja, ele será saída ou entrada

da órbita. Por exemplo, supomos a expressão regular $E = (ab)^*c$, a palavra original $w = ababc$ e a nova palavra $w' = abnabc$, o grafo modificado nos leva as expressões $(abn!)^*c$ e $(n!ab)^*c$, o símbolo ! será usado para representar ambos ? e $*$, assim, a expressão $(abn!)^*c$ representa $(abn?)^*c$ e $(abn^*)^*c$.

As condições e modificações quando for aplicado a regra R_2 possuem três casos que podem ser visualizados na Figura 2.7. Estes casos são detalhados a seguir:

1. No primeiro caso $x = s_{nl}$ e $s_{nr} \in Q^+(x)$. O novo estado s_{new} é introduzido como parte opcional de s_{nl}. Por exemplo, se $E = a(b|c)d$, $w = abd$ e $w' = abnd$, então o grafo modificado leva à expressão regular $E' = a(b\ n!|c)d$.

2. O segundo caso é simétrico ao primeiro.

3. O terceiro caso introduz uma nova alternativa na expressão regular. Por exemplo, se $E = a(b|c)?d$, $w = ad$ e $w' = and$ então o grafo modificado leva à expressão regular $a(b|c|n!)?d$.

Para as condições e modificações quando for aplicado a regra R_3 as condições que devem ser verificadas são: (i) $s_{nl} \in Q^-(x)$ e $s_{nr} \in Q^+(x)$ ou (ii) $s_{nl} \in Q^-(x)$ e $s_{nr} \notin Q^+(x)$ ou (iii) $s_{nl} \notin Q^-(x)$ e $s_{nr} \in Q^+(x)$. As condições (ii) e (iii) são similares à segunda condição da Figura 2.6. Três soluções são propostas. Elas consistem em inserir o novo nó antes, depois ou como nó opcional de x. Por exemplo, dado $E = ab?c$, $w = ac$ e $w' = anc$ nós obtemos o grafo resultando nas expressões $an!b?c$, $ab?n!c$ e $a(n!|b?)c$. A Figura 2.8 mostra estas condições apresentadas e seus respectivos resultados.

No processo de redução do grafo, as regras R_1, R_2, R_3 são aplicadas primeiro dentro das órbitas. Durante este processo de redução, cada órbita é reduzida a um único nó contendo uma expressão regular. É então colocado o fecho transitivo nesta expressão regular, após isto, é realizada a inserção de s_{new} na órbita $\mathcal{O}$, de acordo com algumas situações, as quais são apresentadas na Figura 2.9.

Na Figura 2.9 são apresentados três casos que são considerados na inserção de s_{new} em uma determinada órbita $\mathcal{O}$:

Figura 2.7: Condições e modificações para quando for aplicado a Regra R_2

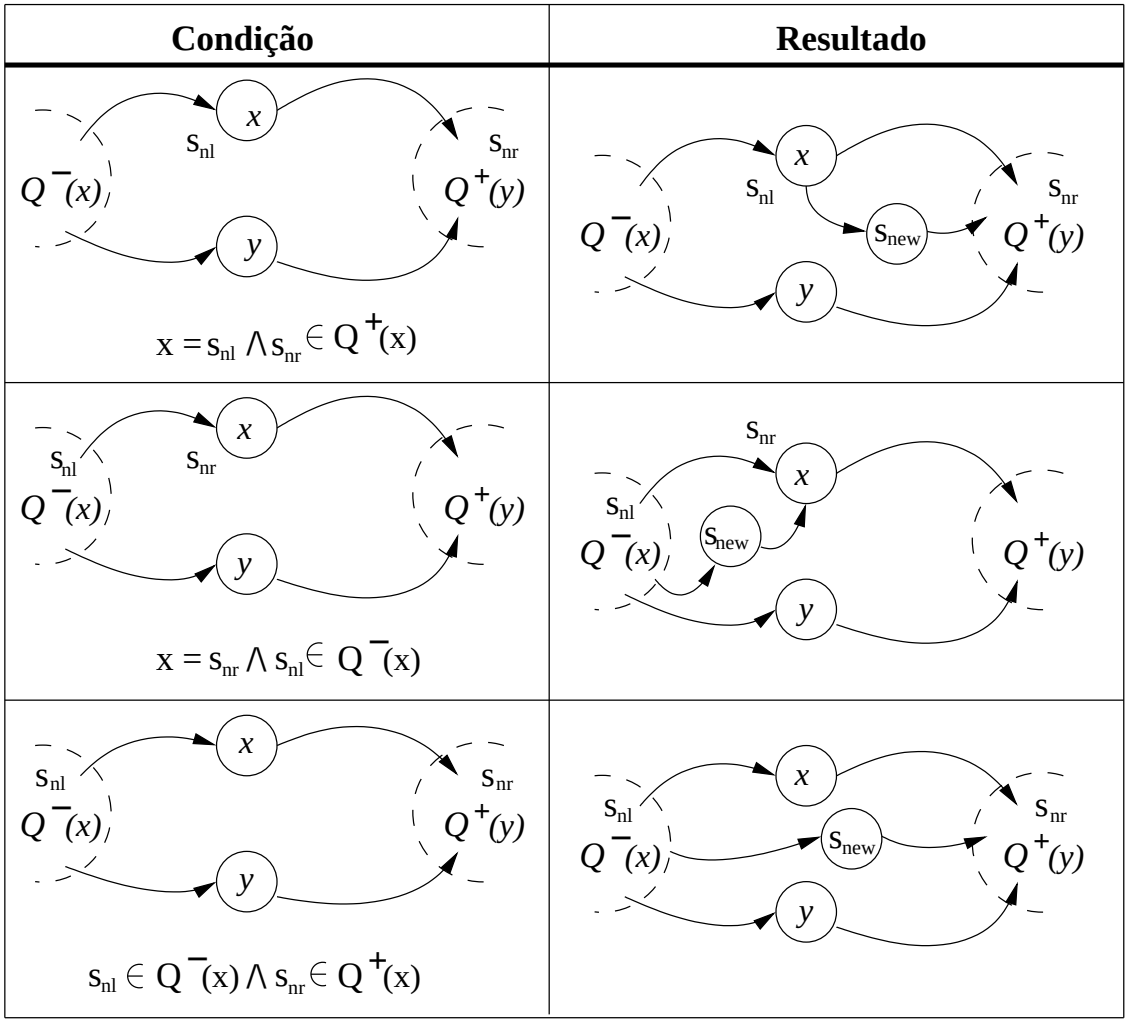
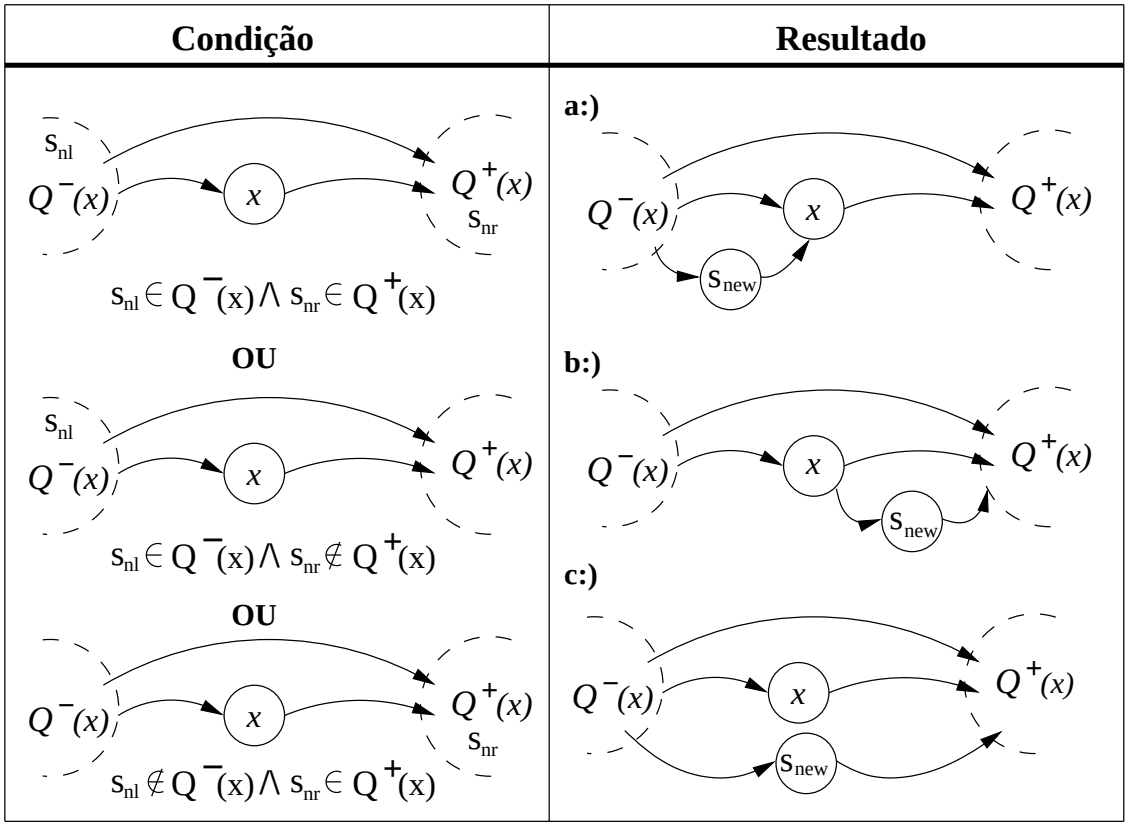


Figura 2.8: Condições e modificações para quando for aplicado a Regra R_3



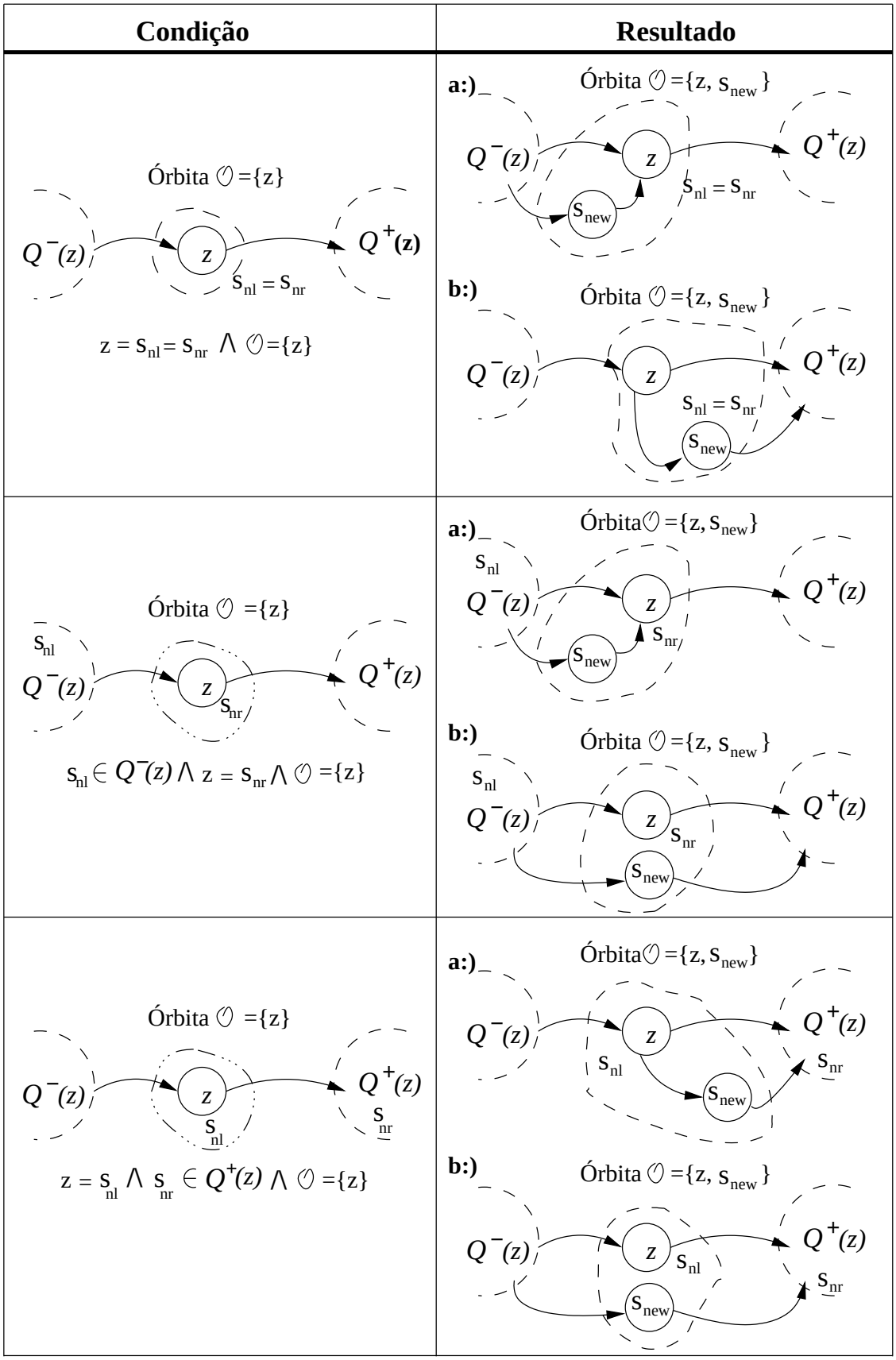
1. No primeiro caso, é verificado se o nó z contém a órbita coincidente com s_{nl} e s_{nr}. Duas soluções são possíveis com inserção de s_{new} no ínicio ou fim da órbita representada por z. Por exemplo, dado $E = a^*$, $w = aa$ e $w' = ana$, o grafo modificado resulta nas expressões $(n!a)^*$ e $(an!)*$.

2. No segundo caso é testado se $s_{nl} \in Q^-(z)$ e $s_{nr} = z$. Aqui, duas soluções são propostas. A primeira é similar à solução proposta no primeiro caso. Na segunda é construída a órbita sobre uma opção entre z e s_{new}. Por exemplo, seja $E = ab^*$, $w = abb$ e w'= $anbb$, nós obtemos a nova expressão regular $E' = a(n!b)^*$ e $E' = a(n?b)^*$.

3. O terceiro caso é simétrico ao segundo.

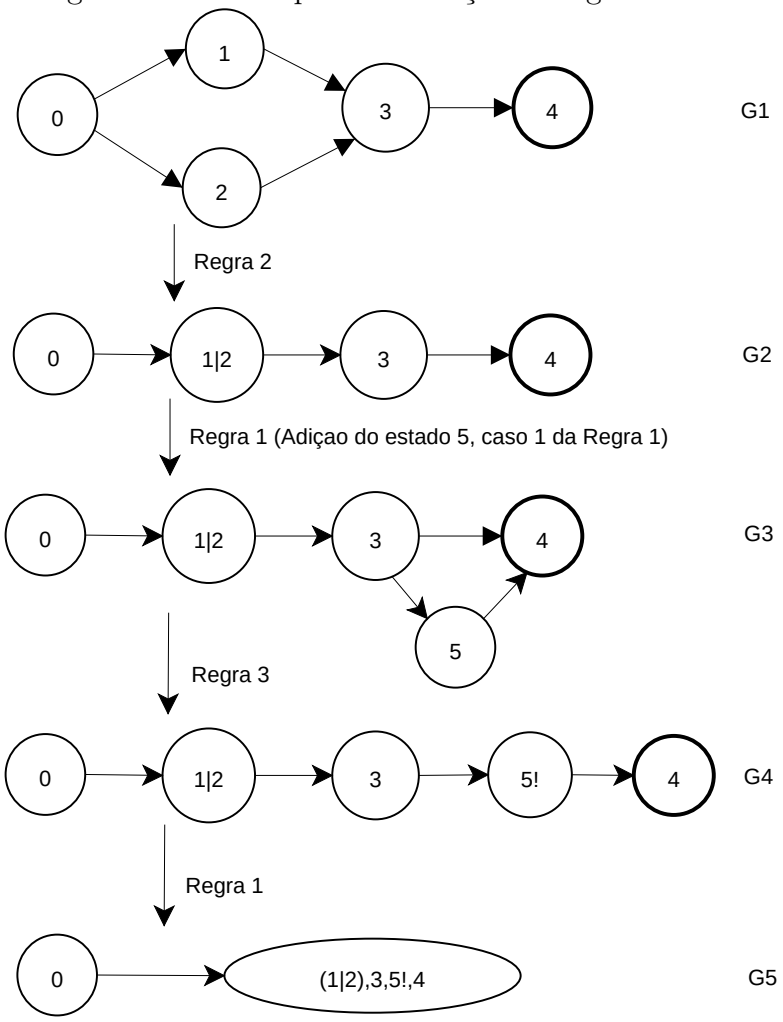
Exemplo 6 (Evolução de expressão regular usando `GREC`) *Dada a expressão regular* $E = (a|b), c, a$, *e a expressão regular indexada* $\bar{E} = (a_1|b_2), c_3, a_4$. *Considerando a palavra* $w = aca \in L(E)$, *que é reconhecida por* E, *e seja a palavra* $w' = acna$, *igual à palavra* w *com a inserção do símbolo* n, *onde* $w' \notin L(E)$.

Figura 2.9: Modificações no grafo da redução de uma órbita



Seja o grafo $G1$ (Figura 2.10), obtido pela transformação da expressão regular E, como apresentado na Seção 2.3. O algoritmo **GREC** *é executado da mesma forma do processo de transformação de grafo em expressão regular (Seção 2.4).*

Figura 2.10: Exemplo de execução do algoritmo GREC



Considerando a execução de **GREC** *em E e na palavra w'. A partir da falha de reconhecimento do grafo $G1$ pela palavra w', são obtidos os estados $s_{nl} = 3$ e $s_{nr} = 4$. Inicialmente, a Regra 2 do processo de redução de grafos (Seção 2.4) pode ser aplicada em $G1$, resultando no grafo $G2$ (Figura 2.10). Agora a regra 1 pode ser aplicada em $G2$, nos vértices $x = 3$ e $y = 4$. Olhando as condições e modificações para quando for aplicado a regra 1, apresentadas na Figura 2.6, claramente o caso 1 se aplica, pois $s_{nl} = x$ e $s_{nr} = y$. Neste caso,* **GREC** *adiciona um novo vértice s_{new} no grafo correspondente ao símbolo n, com posição 5. Transições de x para s_{new} e de s_{new} para y são adicionadas no*

grafo. O novo grafo gerado é G3 (Figura 2.10).

A regra 3, agora pode ser aplicada em G3, nos vértices $3, 4, 5$*, resultando no grafo G4 (Figura 2.10). E aplicando três vezes a regra 1 em G4, é gerado grafo G5, onde existe o vértice com a expressão regular* $E' = (1|2), 3, 5!, 4$*. Substituíndo as posições em* E' *pelos seus símbolos correspondentes temos* $E' = (a|b), c, n!, a$*. Expandindo o símbolo ! em* E'*, temos as duas novas expressões regulares geradas pelo algoritmo* $E'_1 = (a|b), c, n?, a$ *e* $E'_1 = (a|b), c, n^*, a$*.*

2.6 Autômatos de Árvores Regulares

Nesta seção é apresentada uma visão geral sobre autômatos de árvores. Este tipo de autômato pode ser utilizado para verificação da sintaxe de programas, bancos de dados e esquemas para XML. Estes autômatos reconhecem linguagens formadas por árvores, da mesma forma que as linguagens reconhecidas por autômatos finitos são formadas por palavras.

Existem várias propostas de autômatos de árvore. As mais difundidas implementam uma determinada maneira de percorrer a árvore a ser reconhecida. Neste trabalho, serão utilizados os autômatos de árvore *bottom-up*, mas existem propostas de autômatos *top-down* ou até mesmo mistas.

Um autômato de árvore *top-down* inicia a sua execução a partir do estado inicial, e vai descendo a árvore desde o nodo raiz até os nós folhas. Uma abordagem mais detalhada sobre autômatos de árvores *top-down* pode ser encontrada em [CDG+97].

O autômato *bottom-up* inicia a sua computação pelos nós folhas subindo a árvore até o nó raiz. Em [CDG+97] é definido o autômato de árvore finito *bottom-up* (NFTA) como uma tupla $\mathcal{A} = (Q, F, Q_f, \Delta)$, onde Q é um conjunto finito de estados, F é o alfabeto onde o autômato é definido, $Q_f \subseteq Q$ é um conjunto de estados finais e Δ é um conjunto finito de regras de transição com a seguinte estrutura:

$f(q_1(x_1), ..., q_n(x_n)) \rightarrow q(f(x_1, ..., x_n))$, onde $n \geq 0$, $f \in F_n$, $q, q_1, ..., qn \in Q$ e $x_1, ..., x_n \in \mathcal{X}$.

Em [BDAL03] e [BA03] é apresentada uma extensão do autômato NFTA, o autômato

de árvore finito *bottom-up* não-determinístico extendido (ENFTA), o qual trabalha com atributos e elementos de documentos XML e utiliza a DTD como linguagem de esquema. Neste trabalho, o ENFTA será utilizado para representar documentos DTDs, possibilitando uma maneira de percorrer os mesmos. A seguir é definido o ENFTA:

Definição 2.6.1 *Um Autômato de Árvore Finito Bottom-up Não Determinístico Estendido (ENFTA)* sobre Σ é uma tupla $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ onde Q é um conjunto finito de estados, $Q_f \subseteq Q$ é um conjunto de estados finais e Δ é um conjunto finito de regras de transição na forma $a, S, E \rightarrow q$ onde (i) $a \in \Sigma$; (ii) S é um par de conjuntos disjuntos de estados, *i.e.*, $S = \{S_{comp}, S_{op}\}$ (com $S_{comp} \subseteq Q$ e $S_{op} \subseteq Q$); (iii) E é uma expressão regular sobre Q, e (iv) $q \in Q$.

O autômato de árvore extendido $\mathcal{A}$ difere do autômato regular de árvore na forma das transições das regras. A execução do autômato inicia sua computação pelas folhas e então simultaneamente trabalha subindo os caminhos da árvore. Para mover para cima, para uma posição p na árvore, o autômato tem que verificar se os filhos de p respeitam todas as restrições de atributos e de elementos, impostas pelas regras de transição do autômato.

Para que o autômato ENFTA possa ser executado em um determinado documento XML, o documento precisa ser representado na forma de árvore com a seguinte estrutura: o elemento de nível mais alto é o nodo raiz da árvore e cada elemento tem seus subelementos e atributos como nós filhos. Elementos e atributos associados com um texto qualquer têm um filho chamado de nó de dados, mas o seu valor não é representado na árvore. Todo nó folha é um elemento de dados.

A seguir são apresentadas as definições para árvores que representam os documentos XML. Seja um alfabeto finito Σ e U representa o conjunto N^* de todas as palavras finitas incluindo a palavra vazia ϵ.

Definição 2.6.2 *Relação de prefixo:* A relação prefixa em U, denotada por $\preceq$ é definida por: $u \preceq v$ iff $uw = v$ para qualquer $w \in U$. Considerando um subconjunto finito $D \subseteq U$. Ou seja, D é fechado sob prefixos (*closed under prefixes*) se $u \preceq v$, $v \subseteq D$ implica $u \subseteq D$

Definição 2.6.3 *Árvore Σ-rotulada t:* Uma árvore Σ-rotulada t (ou simplesmente uma

árvore) é um mapeamento $t : dom(t) \Rightarrow \Sigma \cup \{\lambda\}$ onde $dom(t) \subseteq U$ é um conjunto não vazio tal que seus prefixos satisfazem: $j \geq 0$, $uj \in dom(t)$, $0 \leq i \leq j \rightarrow ui \in dom(t)$ e onde λ é um símbolo especial que indica a árvore vazia. Uma árvore vazia t tem $dom(t) = \{\epsilon\}$ e $t(\epsilon) = \lambda$. O conjunto $dom(t)$ é também chamado de conjunto de posições de t.

Definição 2.6.4 *Transformação de DTDs em Autômatos de Árvores:* Dada uma DTD d, a transformação de d em um autômato de árvore estendido $\mathcal{A}_d = (Q, \Sigma, Q_f, \Delta)$ é realizada de acordo com os seguintes passos:

- $Q = \{q_a | a \in \Sigma\}$.
- O alfabeto Σ é igual ao alfabeto de d.
- Q_f é um conjunto de estados finais.
- O conjunto de regras de transição Δ é construído da seguinte maneira:

1. Para cada declaração de *elemento* em d na forma `<!ELEMENT eleName contentModel>` é construída uma nova regra de transição $a, S, E \rightarrow q_a$ onde:
   - $S = \{\emptyset, \emptyset\}$;
   - O símbolo a é o nome do elemento $eleName$ e assim q_a é o estado correspondente a a;
   - A expressão regular E, presente na regra de transição, é definida em função do conteúdo do elemento. O conteúdo do elemento $contentModel$ é definido como:
     - Se o conteúdo do elemento $contentModel$ for uma expressão regular $regExp$, então E é a expressão regular obtida pela troca de todos os a em $regExp$ por q_a.
     - Se $contentModel =$ `#PCDATA` então $E = q_{data}$.
     - Se $contentModel =$ `EMPTY` então $E = \emptyset$.

2. Para cada declaração de *atributo* em d na forma `<!ATTLIST eleName attSet>`, onde *attSet* é um conjunto de atributos definido como $attSet = \{at_1, ..., at_n\}$, com $n \geq 0$, e cada atributo *at* em *attSet* é definido como $at = \{$*att-name*,*att-kind*,*att-status*$\}$ onde *att-name* é o nome do atributo, *att-kind* é o tipo do atributo, e *att-status* é o *status* do atributo. Considerando cada atributo *at* em *attSet* faça:

   (a) Construir uma nova regra de transição $att,\{\emptyset,\emptyset\},q_{data} \rightarrow q_{att}$, tal que $att =$ *att-name* de *at*.

   (b) Alterar o conjunto S na regra de transição $eleName,S,E \rightarrow q_{eleName}$ como segue: Se *att-status* = `#REQUIRED` então $S_{comp} = S_{comp} \cup \{q_{att-name}\}$ caso contrário $S_{op} = S_{op} \cup \{q_{att-name}\}$.

3. Incluir a regra $data,\{\emptyset,\emptyset\},\emptyset \rightarrow q_{data}$ em Δ.

A transformação do DTD $\mathcal{D}$ em um autômato de árvore extendido é realizada através do algoritmo descrito na Definição 2.6.4. Esta transformação é realizada em cada regra presente no DTD do Exemplo 1 (clientes.dtd) da Seção 2.1.2 e é detalhada a seguir:

1. **Regra** (`<!ELEMENT clientes (cliente*)>`): Esta é uma regra de elemento e como o conteúdo do elemento é uma expressão regular, a regra de transição é contruída substituindo `cliente`* por $q_{cliente}$`*`, resultando na regra de transição 1.

2. **Regra** (`<!ELEMENT cliente (nome,endereco,e-mail)>`): Este caso é igual ao primeiro, com a substituição da expressão regular `(nome endereco e-mail)` por (q_{nome} $q_{endereco}$ q_{e-mail}), resultando na regra de transição 2 (cliente, $\{\emptyset, \emptyset\}$, q_{nome} $q_{endereco}$ $q_{e-mail} \rightarrow q_{cliente}$).

3. **Regra** (`<!ELEMENT nome (#PCDATA)>`): Esta regra também é uma restrição de elemento, e agora o conteúdo do elemento é `#PCDATA`, então a expressão regular E da nova regra de transição assume q_{data}, resultando na regra de transição 3.

4. **Regra** (`<!ELEMENT endereco (#PCDATA)>`): Igual ao item 3, resultando na regra de transição 4.

5. **Regra** (`<!ELEMENT e-mail (#PCDATA)>`): Igual ao item 3, resultando na regra de transição 5.

6. **Regra** (`<!ATTLIST cliente id ID #REQUIRED>`): Agora a restrição é de atributo. Neste caso uma nova regra de transição é criada para representar o atributo *id*, que pode ser visualizada na regra 6 do exemplo; e deve ser realizada a alteração no conjunto S da regra de transição que representa o elemento correspondente ao atributo, neste caso na regra 2. Como o atributo contém a palavra reservada `#REQUIRED`, ou seja, ele não é obrigatório, então uma referência a ele é colocada no subconjunto S_{comp} de S, assim, $S = \{q_{id}, \emptyset\}$. A regra alterada do elemento *cliente* pode ser visualizada na regra de transição 2.

7. Ao final do processo é adicionada a regra (data, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{data}$), que representa os nós de dados da árvore XML.

O autômato de árvore $\mathcal{A}$ obtido a partir da transformação de $\mathcal{D}$ pode ser visualizado no Exemplo 7.

Exemplo 7 (Autômato $\mathcal{A}$ correspondente a DTD $\mathcal{D}$)

(1) clientes, $\{\emptyset, \emptyset\}$, $q_{cliente}* \rightarrow q_{clientes}$
(2) cliente, $\{\{q_{id}\}, \emptyset\}$, $q_{nome}\ q_{endereco}\ q_{e-mail} \rightarrow q_{cliente}$
(3) nome, $\{\emptyset, \emptyset\}$, $q_{data} \rightarrow q_{nome}$
(4) endereco, $\{\emptyset, \emptyset\}$, $q_{data} \rightarrow q_{endereco}$
(5) e-mail, $\{\emptyset,\emptyset\}$, $q_{data} \rightarrow q_{e-mail}$
(6) id, $\{\emptyset, \emptyset\}$, $q_{data} \rightarrow q_{id}$
(7) data, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{data}$

Dada uma regra r pertencente a um autômato de árvore $\mathcal{A}$, a função $rule\_type(r)$ define se a regra r é do tipo *element*, *attribute* ou *data*:

$$rule\_type(r = a, S, E \rightarrow q_a) = \begin{cases} data & \text{se } a = data \\ attribute & \text{se } a_0 = @ \\ element & \text{caso contrário} \end{cases}$$

Definição 2.6.5 *Transformação de Autômato de Árvore em DTD :* Dado um autômato de árvore $\mathcal{A}$, a transformação de $\mathcal{A}$ em um documento DTD D é realizada da seguinte

forma: Para cada regra $r = a, S, E \rightarrow q_a$, onde $rule\_type(r) = element$, presente no autômato de árvore $\mathcal{A}$ faça:

- Adicione a regra na DTD D: `<!ELEMENT` a (E)`>`
- Para cada attributo $att \in S_{op}$ ($S_{op} \in S$), adicione a seguinte regra da DTD D: `<!ATTLIST` a att`>`
- Para cada attributo $att \in S_{comp}$ ($S_{comp} \in S$), adicione a seguinte regra da DTD D: `<!ATTLIST` a att `#REQUIRED>`

Definição 2.6.6 *Árvore XML $\mathcal{T}$:* Seja $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ um alfabeto onde Σ_{ele} é o conjunto dos nomes dos elementos e Σ_{att} é o conjunto dos nomes dos atributos. Uma árvore XML é definida como uma tupla $\mathcal{T} = (t, type)$, onde t é uma árvore Σ-rotulada e $type$ é uma função definida como a seguir, com $p \in dom(t)$.

$$type(t,p) = \begin{cases} data & se\ t(p) = data \\ element & se\ t(p) \in \Sigma_{ele} \\ attribute & se\ t(p) \in \Sigma_{att} \end{cases}$$

Em [BA03] é necessário o tratamento dos valores de cada nó folha da árvore, ou seja, os dados presentes nos documentos. Com isto, as definições abordam tanto a estrutura quanto os próprios dados dos documentos XML. No contexto deste trabalho somente a estrutura do documento é necessária. Consequentemente, somente definições sobre a estrutura dos documentos serão utilizadas.

Dada uma árvore XML t, o conjunto $posEle$ é o conjunto de todas as posições p em t com $type(t,p) = element$, e o conjunto $posAtt$ é o conjunto de todas as posições p em t com $type(t,p) = attribute$. Considerando a execução de um ENFTA $\mathcal{A}$ em t. Para assumir um estado q de $\mathcal{A}$ na posição p da árvore t, o autômato $\mathcal{A}$ executa os seguintes testes:

1. Se p tem filhos pertencentes ao conjunto $posAtt$ (atributos), a verificação realizada é se o estado na posição p pertence ao conjunto S, o qual é representado por dois tipos: S_{comp} que caracteriza que o estado pertencente a este tipo deve ser obrigatório, e S_{op} indica que os estados pertencentes a este grupo são opcionais.

2. Se p tem filhos pertencentes ao conjunto $posEle$ (elementos) então o teste é realizado por meio de um autômato finito m_E, definido por uma expressão regular E, o qual tenta reconhecer a concatenação dos p' estados filhos.

Definição 2.6.7 *Execução de um Autômato de Árvore ($\mathcal{A}$) em uma Árvore Finita t* : Seja t uma árvore e $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ um ENFTA. A execução de $\mathcal{A}$ em t é uma árvore r tal que $dom(r) = dom(t)$ definida da seguinte maneira: para cada posição p onde seus filhos estão nas posições $p0, ..., p(n-1)$ (com $n/ \geq 0$), o estado q é assumido na posição p, ou seja, $r(p) = q$ se todas as seguintes condições são satisfeitas:

1. $t(p) = a \in \Sigma$

2. Existe uma transição $a, S, E \rightarrow q$ em Δ tal que E é uma expressão regular correspondente ao autômato finito m_E.

3. Existe um inteiro $0 \leq i \leq (n-1)$, tal que os filhos de p podem ser classificados da seguinte forma:

   (a) as posições $p0, ..., p(i-1)$ são membros do conjunto $posAtt$, ou seja, são posições correspondentes a atributos, e

   (b) as posições $pi, ..., p(n-1)$ são membros do conjunto $posEle$, ou seja, são posições correspondentes a elementos, e

   (c) cada filho de p faz parte ou de $posAtt$ ou de $posEle$ e nunca ambos.

4. A árvore r é sempre definida pelos p' filhos, isto é, $r(p0) = q_0, ..., r(p(n-1)) = q_{n-1}$.

5. A palavra $q_i ... q_{n-1}$, composta pela concatenação dos estados associados com as posições em $posEle$, pertence à linguagem gerada por E.

6. Os conjuntos de S respeitam as seguintes propriedades:

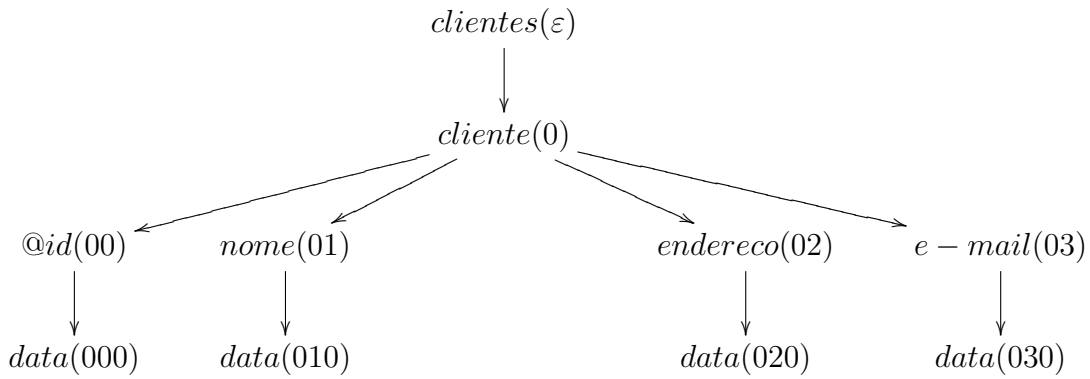
   (a) $S_{comp} \subseteq \{q_0, ..., q_{i-1}\}$ onde $\{q_0, ..., q_{i-1}\}$ é o conjunto de todos os estados associados com as posições i em $posAtt$ e

   (b) $(\{q_0, ..., q_{i-1}\}\ S_{comp} \subseteq S_{op}$.

A execução r é bem sucedida se o estado final for $r(\epsilon)$, ou seja, se for possível executar as operações de movimento dos nós folhas até o nó raiz. Então a árvore t é terminada com um estado de sucesso.

Sejam $\mathcal{X}$ um documento XML e $\mathcal{D}$ uma DTD, que são apresentados respectivamente nos Exemplos 2 (clientes.xml) e 1 (clientes.dtd) da Seção 2.1.2. Dado autômato de árvore $\mathcal{A}$ obtido a partir da transformação de $\mathcal{D}$ apresentado no Exemplo 7, e a árvore XML $\mathcal{T}$, presente no Exemplo 8, que representa o documento XML $\mathcal{X}$.

Exemplo 8 (Árvore XML $\mathcal{T}$ correspondente ao documento $\mathcal{X}$)



O autômato $\mathcal{A}$ é executado na árvore $\mathcal{T}$ no sentido *bottom-up*, ou seja, iniciando pelos nós folhas, que no caso dos autômatos de árvores, oriundos da tradução a partir de DTDs, sempre são os nós rotulados com *data*. Em $\mathcal{T}$ os nós folhas são os estados com as possições 000, 010, 020 e 030. A execução de $\mathcal{A}$ em $\mathcal{T}$ é detalhada a seguir:

1. a regra 7 do autômato $\mathcal{A}$ é aplicada nos nós 000, 010, 020 e 030, sendo que cada um destes nós assume o estado de q_{data};

2. acima de cada um dos nós onde foram aplicados a regra 7 estão os nós 00(@*id*), 01(*nome*), 02(*endereco*) e 03(*e-mail*) respectivamente, como as regras destes nós 3, 4, 5, 6 exigem um estado q_{data}, o qual ja foi verificado, o autômato continua sua execução, agora aplicando as regras 3, 4, 5 e 6 respectivamente nas posições 00, 01, 02, 03, assumindo os estados q_{nome}, $q_{endereco}$, q_{email} e q_{id}, respectivamente.

3. o próximo estado acima das posições 00, 01, 02 e 03 é $0(cliente)$, o qual corresponde à regra 2 do autômato. Esta regra possui como restrição de elementos a expressão regular $q_{nome}\ q_{endereco}\ q_{e-mail}$, e portanto, para que esta restrição seja satisfeita, os filhos de $0(cliente)$, que são elementos, devem satisfazer esta expressão. A verificação é realizada a partir de um autômato finito m_E correspondente a E. Executanto m_E nos filhos elementos de $0(cliente)$, claramente esta execução é realizada com sucesso, portanto esta restrição é satisfeita. Além da restrição de elemento, existe a restrição do atributo q_{id}, que como ele está no conjunto S_{comp}, ele é obrigatório, portanto deve existir um filho de $0(cliente)$ que seja atributo e que seja rotulado com q_{id}, claramente existe este filho, portanto esta condição também é satisfeita. Como todas as condições presentes na regra 2 são satisfeitas, a posição 0 assume o estado $q_{cliente}$;

4. o nó que está acima de $00(cliente)$ é $\varepsilon(clientes)$, e tem como restrição de elemento a expressão regular $q_{cliente}{}^*$, ou seja, aceita vários $q_{cliente}$ como filho. Como seu unico filho existente é $0(q_{cliente})$, esta condição é satisfeita, portanto o estado assumido pelo autômato é ε, sendo assim a execução de $\mathcal{A}$ em $\mathcal{T}$ foi bem sucedida.

Os autômatos de árvore são uma forma eficiente de trabalhar na validação e de percorrer documentos XML a partir de DTDs correspondentes a estes documentos.

CAPÍTULO 3

EVOLUÇÃO INCREMENTAL DE ESQUEMAS PARA XML

Atualmente, muitas organizações utilizam documentos XML para armazenar suas informações. Uma grande quantidade de aplicações utilizam estes documentos para compartilhar dados. A linguagem XML é muito difundida, principalmente para o compartilhamento de dados, onde o ambiente é altamente hetereogêneo, possuindo diversas arquiteturas de *hardware* e de *software*. XML é independente de arquitetura, sendo a sua especificação aberta.

Por exemplo, se considerarmos um ambiente de vendas, onde são armazenados dados sobre todas as transações que envolvem este negócio, claramente a quantidade de informação que deve ser armazenada pode ser muito grande.

Caso a escolha do meio de armazenamento seja documentos XML, provavelmente existirá uma grande quandidade de documentos, os quais para serem bem organizados, precisam de esquemas. Suponhamos que seja necessária uma alteração na estrutura destes documentos, para que comporte novos campos, necessários ao negócio, mantendo a compatibilidade com documentos anteriores, válidos em relação ao esquema original. Uma possível solução, seria alterar todos os documentos existentes, de forma a deixá-los de acordo com as novas alterações. O problema é que esta solução pode ter um custo muito alto, se a quantidade de documentos for muito grande. Outra solução é alterar somente o esquema XML, de maneira que, o mesmo continue reconhecendo os arquivos ja existentes, e que também comporte as novas alterações.

Por exemplo, suponha que as informações que são gravadas de um cliente sejam: *id* (identificador do cliente), *nome* (nome do cliente), *endereco* e *e-mail*, e que existe uma grande quantidade de documentos XML que possuem esta estrutura (no Exemplo 1 da Seção 2.1.2 é possível visualizar um documento DTD que corresponde a essa estrutura). Suponhamos que exista a necessidade, para novos dados, de adicionar o campo *telefone*.

Então uma alteração no esquema é proposta para que o mesmo possa aceitar tanto os documentos antigos, que não possuem o campo *telefone*, como os dados novos, que possuem o campo *telefone*.

Claramente, como esta alteração somente é realizada no esquema, ela é mais eficiente do que alterar todos os documentos antigos. Neste trabalho é apresentada uma solução para criar este novo esquema. Chamamos este problema de evolução incremental no esquema XML, visto que, esta evolução deve ser conservativa, ou seja, não perdendo a sua generalidade em relação aos documentos antigos.

Em [GMR05] é apresentado um algoritmo para trabalhar com o problema de evolução de esquemas para XML. É proposto um conjunto de evoluções primitivas, que podem ser aplicadas em componentes básicos de um esquema, tais comos, elementos e tipos. Este conjunto de evoluções primitivas é completo, no sentido de que todas as transformações possíveis podem ser expressadas através de um sequência de elementos deste conjunto.

Um conjunto de alto nível de modificações é proposto, os quais são formados por sequências de alterações primitivas. Este conjunto de alto nível é construído de acordo com as necessidades gerais de modificações. Eles facilitam a representação das modificações mais usuais.

O algoritmo apresentado em [GMR05] trabalha com evoluções em esquemas para XML do tipo *XML Schema*.

A idéia basica do algoritmo apresentado em [GMR05] é manter as partes do esquema que correspondem a partes dos documentos XML inalteradas, sem alterações, alterando somente partes do esquema que correspondam às alterações realizadas nos documentos XML. As partes dos documentos afetadas por estas modificações podem ser identificadas e revalidadas, assim diminuíndo o custo do processamento.

Três categorias de alterações primitivas são apresentadas: inserção, modificação e exclução de componentes de um esquema XML. As modificações são classificadas em três categorias: estrutural, renomeação, e modificações de migração.

Modificações estruturais permitem modificar o tipo de um elemento e suas restrições. Renomeações permitem renomear o nome de um elemento ou tipo. Modificações de mi-

grações incluem: mover um sub-elemento de um elemento para outro e transformar um elemento ou tipo de escopo local para um escopo global, e vice-versa.

Modificações de alto nível podem ser compostas usando modificações primitivas para expressar de uma maneira mais compacta modificações compactas correspondentes a necessidades comuns de evolução. Suas aplicações permiteum executar sequências de modificações primitivas.

Já em [Kra00], Kramer propõe a *framework* XEM (*XML Evolution Mananager*), para trabalhar com a evolução de DTDs e documentos XML. XEM disponibiliza um conjunto básico de alterações nos dados e nos esquemas.

Para alterações de dados, o algoritmo garante que os documentos XML alterados continuem respeitando a estrutura e restrições impostas pelo DTD. E para alterações no esquema, ele assegura que o novo DTD seja bem formado, e que todos os documentos XML sejam transformados para estarem de acordo com o novo DTD.

As operações de alterações são divididas em duas categorias: as que correspondem a documentos XML e as que correspondem a esquemas DTD.

Utiliza algumas funções de alteração de documentos DTD tais como:
$insertDTDElement(E, i, q, d)$ que insere um elemento E na posição i relativa ao nó q e com o valor padrão d, e $removeDTDElement(E, i, q, d)$ que remove o elemento na posição i correspondente ao nó E.

Também utiliza algumas funções de alterações em documento XML tais como:
$addDataElement(de, i)$ que adiciona um elemento de na posição i, e
$destroyDataElement(de, i)$ que remove o elemento de da posição i do documento XML.

A idéia do algoritmo apresentado em [Kra00], é quando alguma modificação for feita em algum documento XML, modificações são realizadas no documento DTD correspondente a este documento XML, para satisfazer as novas necessidades, e também nos outros documentos XML para que os mesmos continuem sendo aceitos pelo novo documento DTD.

Diferentemente das duas abordagens apresentadas anteriormente, o nosso algoritmo de evolução de esquemas, apresentado na próxima seção, mantém a generalidade do novo

esquema com todos os documentos antigos, sem a necessidade de alterações nos mesmos, atendendo também as novas necessidades, expressas pelas modificações realizadas em um documento XML.

3.1 Algoritmo de Evolução Incremental de Esquemas para XML

Inicialmente, os documentos XML são representados na forma de árvores rotuladas, como apresentado na Seção 2.6 sobre Autômatos de Árvores Regulares. Cada elemento e atributo do documento corresponde a um nó rotulado na árvore XML. Este rótulo corresponde à posição em que o nó se encontra na árvore. Os esquemas XML são representados na forma de Autômato de Árvore, como também apresentado na Seção 2.6, um algoritmo é utilizado para realizar esta transformação. Toda a computação que deve ser realizada no esquema, é realizada sobre o autômato de árvore correspondente a este esquema. A seguir é apresentada uma solução para evolução incremental de esquemas para XML.

Sejam $\mathcal{D}$ um esquema XML, $X \in L(\mathcal{D})$ um documento XML que satisfaz as restrições do esquema $\mathcal{D}$, e seja $X' \notin L(\mathcal{D})$ um novo documento gerado a partir do documento X com algumas modificações. A função *ISE* [1], definida a seguir, gera os novos esquemas $\mathcal{D}'$ que aceitam os dois documentos X e X' $(X, X' \in \mathcal{D}')$.

Definição 3.1.1 *Alteração μ em uma árvore XML $\mathcal{T}$:* Seja $\mathcal{T}$ uma árvore XML. E seja μ uma única modificação em $\mathcal{T}$. Esta modificação pode ser de elemento ou de atributo. Uma modificação μ pode assumir um dos seguintes casos:

- $insert\_element(t, t', pj)$: corresponde à inserção de uma subárvore t' de elementos na posição pj da árvore original t;
- $delete\_element(t, pj)$: corresponde à exclusão de um elemento na posição pj na árvore t.
- $insert\_attribute(t, att, pj)$: corresponde à inserção de um atributo att na posição pj da árvore t;

[1] *ISE* (*Incremental Schemas Evolution*), Algoritmo de Evolução Incremental de Esquemas para XML.

- *delete_attribute*(t, pj): corresponde à exclusão de um atributo na posição pj na árvore t.

As modificações realizadas no documento XML original, que serão utilizadas para a evolução do esquema, são representadas na forma de lista. Algumas restrições são utilizadas para definir como esta lista deve ser formada, isto para tornar o processamento do algoritmo mais eficiente. A seguir é apresentada uma definição para esta lista de modificações, bem como, as restrições de como ela deve ser formada.

Definição 3.1.2 *Lista de modificações L em uma árvore XML $\mathcal{T}$:* Seja $\mathcal{T}$ uma árvore XML, L é uma lista de modificações na forma $L = \{\mu_0, ..., \mu_{n-1}\}$, com $n \geq 0$, onde μ é uma modificação na árvore XML. Um lista L de alterações possui as seguintes restrições:

- Uma alteração na posição p exclui alterações em descendentes de p. Em outras palavras, não existem em L duas posições p e p' tal que $p \preceq p'$;
- Se uma alteração μ aparece mais de uma vez em L então a alteração μ envolvendo p é de inserção.
- Alterações em L são ordenadas pela posição p, de acordo com a ordem do documento.
- Uma posição de alteração em L sempre se refere à árvore original.

A seguir são apresentadas algumas funções utilizadas no algoritmo de evolução de esquemas para XML:

- *node*(p, t) retorna o nó que está na posição p da árvore XML t;
- *children*$(node)$ retorna um conjunto de todos os nós presentes na árvore XML que são filhos de *node*.
- *parent*$(node)$ retorna o nó pai do nó *node* na árvore XML.
- $elementName(node)$ retorna o nome do nó *node*.
- *concat_elements*$(nodes)$ concatena o nome de todos os nós $\{node \in nodes | nodes = \{node_0, \ldots, node_{n-1}\}\}$, onde n é o número de nós presente em *nodes*.

- $rule(f, \mathcal{A})$ retorna a regra, na forma $a, S, E \rightarrow q$, presente no autômato de árvore $\mathcal{A}$, onde $f = a$;

- $expressionOf(r)$ retorna a expressão regular E presente na regra r da forma $a, S, E \rightarrow q$.

- $changeExp(E, E', r)$ troca a expressão regular E presente na regra $r = a, S, E \rightarrow q$ pela expressão regular E', *i.e.*, $r = a, S, E' \rightarrow q$.

Definição 3.1.3 $ISE(\mathcal{A}, t, L)$ é o conjunto de autômatos de árvore obtidos pela evolução do autômato de árvore $\mathcal{A}$, que representa um documento DTD $\mathcal{D}$ qualquer, com relação à lista de modificações L realizadas no documento XML X, representado pela árvore XML t, onde $X \in L(\mathcal{D})$. $ISE(\mathcal{A}, t, L)$ é computada pelo Algoritmo1. (página 65).

A função $ISE(\mathcal{A}, t, L))$ é executada da seguinte maneira: Primeiramente é criada a árvore XML $\mathcal{T}$ a partir do documento XML X e o autômato de árvore $\mathcal{A}$ a partir do DTD $\mathcal{D}$. Para cada modificação μ em L o algoritmo verifica qual é o tipo desta modificação, e de acordo com este tipo é executado um dos seguintes passos:

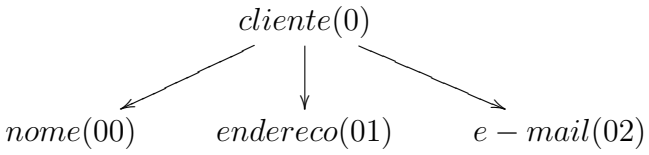
- As modificações do tipo $insert\_element(t, t', pj)$ são agrupadas de acordo com a posição p, ou seja, as modificações que possuem o mesmo nó pai na arvore XML. Seja $insert\_elements$ um conjunto finito das modificações $insert\_element(t, t', pj) \in L$ que possuem a mesma posição p. Seja w a palavra formada pela concatenação de todos os nós filhos, que são nós do tipo elemento, do nó que está na posição p da árvore t. A palavra w', inicialmente é inicializada com o mesmo valor de w, para cada modificação $insert\_element(t, t', pj) \in L$, a palavra n, correspondente ao nó raiz de t', é inserida na posição j de w'.

  Seja f o nó que está localizado na posição p da árvore t. Seja $r \in \mathcal{A}$ a regra, correpondente ao nó f. No processo de inserção de elementos, a expressão regular E presente em r é evoluída utilizando o algoritmo *Evolution-e*, descrito na Seção 4.3. Claramente, temos que a palavra $w \in L(E)$, e $w' \notin L(E)$. O objetivo aqui, é gerar

novas expressões regulares E', tais que, $w, w' \in L(E')$. A geração destas expressões regulares E' é realizada da seguinte maneira: $E' = \mathit{Evolution\text{-}e}(E, w')$.

- Ao final, a expressão regular E presente em r é substituída pelas expressões E', onde o usuário pode escolher a melhor expressão que se adequa às suas necessidades. As expressões regulares E' correspondem à ligação do DTD D_t, correspondente à árvore t, no autômato de árvore $\mathcal{A}$. As outras regras pertencentes ao D_t são adicionadas no autômato $\mathcal{A}$.

  Exemplo: Dada uma regra $r = (\mathit{cliente}, \{\emptyset, \emptyset\}, q_{\mathit{nome}}\ q_{\mathit{endereco}}\ q_{\mathit{e-mail}} \rightarrow q_{\mathit{cliente}})$ pertencente a um autômato de árvore $\mathcal{A}$. Seja a árvore XML t a seguir:

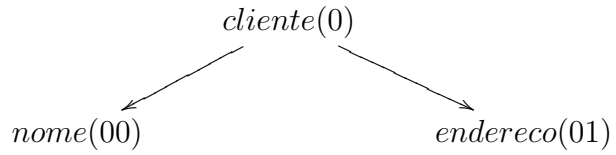


  Dada a lista de modificações $L = \{\mathit{insert\_element}(t, t', 02)\}$ realizadas em t, onde $t' = \mathit{bairro}$ é uma árvore XML com um único elemento. A execução de $\mathit{ISE}(\mathcal{A}, t, L)$ gera um novo autômato $\mathcal{A}'$, onde a expressão regular E presente na regra r foi evoluída, gerando a regra $r' = (\mathit{cliente}, \{\emptyset, \emptyset\}, q_{\mathit{nome}}\ q_{\mathit{endereco}}\ q_{\mathit{e-mail}}\ q_{\mathit{bairro}}? \rightarrow q_{\mathit{cliente}})$ e, a adição da regra $r'' = (\mathit{bairro}, \{\emptyset, \emptyset\}, q_{\mathit{data}} \rightarrow q_{\mathit{bairro}})$.

- Caso a modificação seja $\mathit{delete\_element}(t, pj)$: Seja ele o elemento correspondente à posição pj da árvore t. Seja f o nó pai de ele, ou seja, o nó que está na posição p de t. Seja $r \in \mathcal{A}$ correspondente a f. A alteração que deve ser feita neste caso é na expressão regular E da regra r. O elemento ele presente na expressão regular E deve ser tornado opcional, ou seja, decorado com o símbolo $'?'$. Seja x a posição em E correspondente ao elemento ele. A nova expressão regular E' é gerada por $E' = \mathit{Delete}(E, x)$.

  Exemplo: Dada uma regra $r = (\mathit{cliente}, \{\emptyset, \emptyset\}, q_{\mathit{nome}}\ q_{\mathit{endereco}} \rightarrow q_{\mathit{cliente}})$ perten-

cente a um autômato de árvore $\mathcal{A}$. Seja a árvore XML t a seguir:



Dada a lista de modificações $L = \{delete\_element(t, 01)\}$ realizadas em t. Neste caso, a execução de $ISE(\mathcal{A}, t, L)$ gera um novo autômato $\mathcal{A}'$, onde o estado $q_{endereco}$ correspondente à palavra $endereco(01) \in t$ é tornado como opcional. Desta forma, temos no novo autômato $\mathcal{A}'$, gerado por $ISE(\mathcal{A}, t, L)$, a regra $r = (cliente, \{\emptyset, \emptyset\}, q_{nome}\ q_{endereco}?\ \rightarrow q_{cliente})$

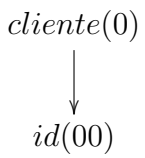
- Caso a modificação seja $insert\_attribute(t, att, pj)$: Seja att o atributo que deve ser adicionado, na forma $att = \{$*att-name*,*att-kind*,*att-status*$\}$. Inicialmente a regra (*att-name*, $\{\emptyset, \emptyset\}$, $q_{data} \rightarrow q_{att-name}$) é adicionada em $\mathcal{A}$. Após é necessário identificar qual regra em $\mathcal{A}$ que possui o elemento com referência a este novo atributo. Isto é feito usando a posição p_j que foi informada. Seja r a regra em $\mathcal{A}$ onde deve ser realizada esta alteração. O que deve ser alterado na regra r é o conjunto S, com $s_{op} = s_{op} \cup q_{att-name}$.

  Exemplo: Dada uma regra $r = (cliente, \{\emptyset, \emptyset\}, \emptyset\ \rightarrow q_{cliente})$ pertencente a um autômato de árvore $\mathcal{A}$. Seja a árvore XML $t = cliente(0)$ com um único elemento.

  Dada a lista de modificações $L = \{insert\_attribute(t, att, 0)\}$ realizadas em t, onde $att = id$. Neste caso, temos no novo autômato $\mathcal{A}'$, gerado por $ISE(\mathcal{A}, t, L)$, com a regra $r = (cliente, \{\emptyset, \{q_{id}\}\}, \emptyset\ \rightarrow q_{cliente})$. A regra $r'' = (id, \{\emptyset, \emptyset\}, q_{data}\ \rightarrow q_{id})$ foi adicionada no autômato $\mathcal{A}'$.

- Caso a modificação seja $delete\_attribute(t, pj)$: Seja att o atributo correspondente à posição pj na árvore t. Seja f o nó que está na posição p da árvore t. Seja r a regra em $\mathcal{A}$ correspondente a f. O processo de exclusão é simples, para isto basta alterar o conjunto S de r, onde $s_{comp} = s_{comp} \setminus att$ e $s_{op} = s_{op} \cup att$, ou seja, remover o atributo att de s_{comp} e adicioná-lo em s_{op}, tornando att como atributo opcional.

Exemplo: Dada uma regra $r = (cliente, \{\{q_{id}\}, \emptyset\}, \emptyset \ \rightarrow q_{cliente})$ pertencente a um autômato de árvore $\mathcal{A}$. Seja a árvore XML t a seguir:



Dada a lista de modificações $L = \{delete\_attribute(t, 00)\}$ realizadas em t. Neste caso, temos no novo autômato $\mathcal{A}'$, gerado por $ISE(\mathcal{A}, t, L)$, onde o estado $\{q_{id} \in s_{comp} \mid s_{comp} \in r\}$ é tornado opcional, removendo-o do conjunto s_{comp} e adicionando-o no grupo $s_{op} \in r$. Desta forma, o autômato $\mathcal{A}'$, gerado por $ISE(\mathcal{A}, t, L)$, passa a ter a regra $r = (cliente, \{\emptyset, \{q_{id}\}\}, \emptyset \ \rightarrow q_{cliente})$.

A função ISE retorna um conjunto de novos autômatos de árvores, representando documentos DTD, que reconhecem o documento XML representado na árvore $\mathcal{T}$ e também todos os documentos que estão de acordo com a lista de modificações L. A partir do conjunto de novos esquemas retornado pelo algoritmo, o usuário pode decidir qual o melhor esquema que se adapta às suas necessidades.

O algoritmo *ISE* demonstra-se ideal para os casos, onde existe a necessidade de evoluir um esquema para XML (DTD), para satisfazer novas necessidades e, que os novos esquemas gerados, mantenham a generalidade com o esquema original. Com a utilização do algoritmo é possivel realizar este trabalho de forma automática, sem a preocupação de perder a generalidade com o esquema original, e com documentos já existentes que são mantidos através deste esquema original. Um trabalho manual pode se tornar exaustivo, dependendo do tamanho e da complexidade do esquema, bem como, conter problemas de generalidade, pois se a complexidade do esquema for muito grande, a chance de erros em alterações manuais são maiores.

Detalhes sobre como o algoritmo *ISE* foi implementado, e alguns testes de performance podem ser encontrados no Capítulo 7.

Algorithm 1 $ISE(\mathcal{A}, t, L)$

function $ISE(\mathcal{A}, t, L)$
  var $inserted\_elements\_pos := \emptyset$
  for each $\mu \in L$ **do**
    if $type(\mu) = insert\_element(t, t', pj) \wedge\ p \notin inserted\_elements\_pos$ **then**
      $w := concat\_elements(children(node(p, t)))$
      $w' := w$
      $insert\_elements := \{insert\_element(t, t', p'j') \in L \wedge\ p' = p\}$
      for each $insert\_element(t'', t''', p'j') \in insert\_elements$ **do**
        $insert\_symbol(w', elementName(root(t'')), j')$
      end for
      $f := node(p, t)$
      $r := rule(f, \mathcal{A})$
      $E := expressionOf(r)$
      $E' := Evolution\text{-}e(E, w')$
      $changeExp(E, E', r)$
      $inserted\_elements\_pos := inserted\_elements\_pos \cup p$
    end if
    if $type(\mu) = delete\_element(t, pj)$ **then**
      $ele := node(pj, t)$
      $f := parent(ele)$
      $r := rule(f, \mathcal{A})$
      $E := expressionOf(r)$
      $x := \{x|\ x \in Pos(E) \wedge\ elementName(ele) = symb(x)\}$
      $E' := Delete(E, x)$
      $changeExp(E, E', r)$
    end if
    if $type(\mu) = insert\_attribute(t, att, pj)$ **then**
      $//att = \{att - name, att - kind, att - status\}$
      $insert\_rule(\mathcal{A}, \text{“}att - name, \{\emptyset, \emptyset\}, q_{data} \rightarrow q_{att-name}\text{”})$
      $f := node(p)$
      $r := rule(f, \mathcal{A})$
      $s := \{s_{op}|\ s_{op} \in r\}$
      $s := s \cup q_{att-name}$
    end if
    if $type(\mu) = delete\_attribute(t, pj)$ **then**
      $att := node(pj, t)$
      $f := node(p, t)$
      $r := rule(f, \mathcal{A})$
      $s_{op} := \{s_{op}|\ s_{op} \in r\}$
      $s_{op} := s_{op} \cup att$
      $s_{comp} := \{s_{comp}|\ s_{comp} \in r\}$
      $s_{comp} := s_{comp} \setminus att$
    end if
  end for
  return $\mathcal{A}$
end function

CAPÍTULO 4

ALGORITMO DGREC

Neste capítulo é apresentado o algoritmo para evolução de expressões regulares dGREC, que é uma otimização do algoritmo GREC, onde a evolução é realizada diretamente na expressão regular, não sendo necessária a transformação da expressão para autômato de Glushokv. Inicialmente, na Seção 4.1 é apresentado algoritmo dGREC, que é uma das contribuições deste trabalho. Uma comparação entre os algoritmos GREC e dGREC é apresentada na segunda seção deste capítulo. Em seguida é apresentada uma extensão do algoritmo dGREC para trabalhar com várias modificações na palavra original. Por fim, é apresentado um exemplo detalhado da execução do algoritmo dGREC.

4.1 Algoritmo dGREC

O algoritmo dGREC [dLM06], [dLFM07] (*Direct Generate Regular Expression Choices*) gera os mesmos resultados que o GREC, porém trabalhando diretamente na expressão regular, dispensando o uso dos autômatos de Glushkov, bem como, a transformação prévia da expressão regular em autômato de Glushkov.

Dada uma expressão regular E, uma palavra w que pertence à linguagem de E ($L(E)$), e a palavra $w' \notin L(E)$ igual a w, com uma modificação, podendo ser uma inserção ou exclusão de um símbolo em w.

O caso da exclusão é simples, basta tornar o símbolo, na posição p de E, correspondente ao símbolo que foi excluído de w, opcional. Este processo é realizado pela definição $Delete(E, p)$, apresentada a seguir:

Definição 4.1.1 $Delete(E, p)$ é a expressão regular obtida pela evolução da expressão E, tornando a posição p opcional em E.

$Delete(E, p)$ pode ser recursivamente computada da seguinte maneira:

$$
\begin{aligned}
Delete(\emptyset, p) &= \emptyset \\
Delete(\varepsilon, p) &= \varepsilon \\
Delete(\alpha_x, p) &= \begin{cases} \alpha_x? & \text{if } x = p \\ \alpha_x & \text{otherwise} \end{cases} \\
Delete(F + G, p) &= \begin{cases} Delete(F, p) + G & \text{if } p \in Pos(F) \\ F + Delete(G, p) & \text{if } p \in Pos(G) \end{cases} \\
Delete(F.G, p) &= \begin{cases} Delete(F, p).G & \text{if } p \in Pos(F) \\ F.Delete(G, p) & \text{if } p \in Pos(G) \end{cases} \\
Delete(F^+, p) &= \begin{cases} F^* & \text{if } |Pos(F)| = 1 \wedge Pos(F) = p \\ Delete(F, p)^+ & \text{otherwise} \end{cases} \\
Delete(F^*, p) &= Delete(F, p)^* \\
Delete(F?, p) &= Delete(F, p)?
\end{aligned}
$$

Dada a expressão regular $E = a_1.b_2.c_3$ e a palavra $w = abc \in L(E)$. Seja $w' = ac$ a palavra w com a remoção do símbolo b_2. A posição do simbolo b na expressão E é $p = 2$. Com a execução de $Delete(E, p)$, temos a nova expressão regular $E' = a_1.b_2?.c_3$, onde o símbolo b_2 foi alterado para opcional na expressão regular. Em um segundo caso, temos $E = a_1.b_2^+.c_3$, $w = abc \in L(E)$ e $w' = ac \notin L(E)$ com a remoção do símbolo b_2 ($p = 2$). Agora $Delete(E, p)$ gera a nova expressão regular $E' = a_1.b_2^*.c_3$, substituindo o símbolo $^+$ por *. □.

Considere agora o caso onde a inserção de um símbolo na palavra $w \in L(E)$ gera uma nova palavra $w' \notin L(E)$. Esta operação de inserção corresponde à inclusão de uma nova posição em E. O objetivo é contruir um algoritmo (chamado *Evolution*) que encontra novas expressões regulares E', tais que, $w' \in L(E')$ e $L(E) \subseteq L(E')$.

Antes da construção do algoritmo, algumas notações usadas no caso da inserção são introduzidas:

(i) Para simplificar a notação, o símbolo ! será utilizado para representar ambos ? e *, assim, a expressão $a.n!$ é uma abreviação para $a.n?$ e $a.n^*$.

(ii) Expressões regulares E sempre terminam com o símbolo #, que representa o final da expressão (*e.g.* , $E = a_1.b_2.\#_3$).

(iii) A função $Update(E, F, G)$ retorna a nova expressão regular E', obtida pela troca da sub-expressão F pela sub-expressão G. Se $E = a_1.b_2.(c_3+d_4)$, $F = (c_3+d_4)$ e $G = g_5?.(c_3+$

d_4), então $Update(E, F, G)$ retorna a nova expressão regular $E' = a_1.b_2.g_5?.(c_3 + d_4)$. Lembrando que, para implementar a função *Update* é necessário encontrar F em E, e então trocar F por G. Diferentes tipos de implementações são possíveis, como discutidas na Seção 4.2.

Dada a expressão regular E e a palavra $w \notin L(E)$, é proposto o algoritmo *Evolution* que executa os seguintes passos:

(1) Compare (em paralelo) a palavra w e a expressão regular E, estabelecendo a correspondência entre as posições em w e em E. Esta comparação continua até ser encontrada uma posição i que não corresponde a uma posição em E. Defina $s_{nl} \in Pos(E)$ a posição em E correspondente à posição $i-1$ em w.

(2) Como i é a posição correspondente ao novo símbolo em w, a palavra original (antes da inserção) era a concatenação das palavras $w[0 : i-1]$ com $w[i+1 : |w|]$. $s_{nr} \in Pos(E)$ é a posição correspondente à posição $i+1$ em w.

(3) Use s_{nl}, s_{nr} e a nova posição s_{new} para computar `dGREC`, resultando no conjunto das novas expressões regulares E'.

Definição 4.1.2 *: Evolution*(E, w) é o conjunto de expressões regulares (obtidas pela evolução da expressão E) que reconhecem as palavras $w \notin L(E)$ e $w \in L(Evolution(E, w))$

Evolution(E, w) é definida pelo Algoritmo 2.

Algorithm 2 *Evolution*(E, w)

```
function Evolution(E, w)
  var R := ∅                                          // Solutions calculated so far.
  var C := {p | p ∈ First(E) and w_0 = symb(p)}        // Possible current positions.
  var i := 0                                          // Current symbol in w.
  var C' ; S                  // Next possible positions and set of triples (s_nl, s_nr, gap).
  for i := 0 to |w| do
    C' := {p' | p' ∈ Follows(E,p) ∧ w_i = symb(p) ∧ w_{i+1} = symb(p') ∧ p ∈ C}
    if C' = ∅ then
      S := {(s_nl, s_nr, gap) | s_nl ∈ C ∧ s_nr ∈ Follows(E, s_nl) ∧ w_{i+gap} = symb(s_nl)}
      for each (s_nl, s_nr, gap) ∈ S do
        R := R ∪ dGREC(E, E, s_nl, s_nr, w[i ... i + gap])
      end for
    end if
    i := i + 1; C := C'
  end for
  return R
end function
```

A principal parte do algoritmo *Evolution* é a computação de dGREC (Definição 4.1.3). Para executar a evolução de uma expressão regular, dGREC considera diferentes tipos de expressões regulares. Para este fim, as funções *evOr*, *evAnd*, *evClosure* e *evOpt* são introduzidas (ver Definições de 4.1.4 a 4.1.7).

Definição 4.1.3 *:* $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, s_{new})$ é o conjunto das expressões regulares E' obtido pela evolução da expressão regular E. Cada expressão E' é obtida a partir de E, pela inserção de uma nova posição s_{new}, de acordo com as posições s_{nl} e s_{nr}. S é uma expressão regular.

$\texttt{dGREC}(S, E, s_{nl}, s_{nr}, s_{new})$ é computado como a seguir:

$$
\begin{array}{lcl}
\texttt{dGREC}(\emptyset, E, s_{nl}, s_{nr}, s_{new}) & = & \emptyset \\
\texttt{dGREC}(\varepsilon, E, s_{nl}, s_{nr}, s_{new}) & = & \emptyset \\
\texttt{dGREC}(\alpha_x, E, s_{nl}, s_{nr}, s_{new}) & = & \emptyset \\
\texttt{dGREC}(F + G, E, s_{nl}, s_{nr}, s_{new}) & = & \mathit{evOr}(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F.G, E, s_{nl}, s_{nr}, s_{new}) & = & \mathit{evAnd}(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F^+, E, s_{nl}, s_{nr}, s_{new}) & = & \mathit{evClosure}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F^*, E, s_{nl}, s_{nr}, s_{new}) & = & \mathit{evClosure}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F?, E, s_{nl}, s_{nr}, s_{new}) & = & \mathit{evOpt}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
 & & \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new})
\end{array}
$$

Definição 4.1.4 $\mathit{evOr}(F, G, E, s_{nl}, s_{nr}, s_{new})$ é o conjunto de expressões regulares obtidas pela evolução da expressão $F+G$ de acordo com as posições s_{nl} e s_{nr}, onde E é a expressão regular original e s_{new} é a posição do novo símbolo a ser inserida em E.

$\mathit{evOr}(F, G, E, s_{nl}, s_{nr}, s_{new})$ é computada pelo Algoritmo 3.

□

O Algoritmo 3 propõe alterações na sub-expressão $F+G$ de E. No primeiro caso do algoritmo, quando $s_{nl} \in \mathit{Last}(F)$ e $s_{nr} \in \mathit{Follow}(E, s_{nl})$, a expressão F é trocada por $F.s_{new}!$. O segundo caso é simétrico ao primeiro, a sub-expressão F é substituída por $s_{new}!.F$. No terceiro caso, uma nova alternativa é introduzida na expressão regular, substituindo G por $G + s_{new}!$. Por exemplo, considerando a expressão $E = a_1.(b_2 + c_3)?.d_4$, $w = ad$ e

Algorithm 3 $evOr(F, G, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                    // Solutions calculated so far.
if s_nl ∈ Last(F) ∧ s_nr ∈ Follow(E, s_nl) then
   R := R ∪ Update(E, F, F.s_new!)                            // Case 1.
end if
if s_nr ∈ First(F) ∧ s_nl ∈ Previous(E, s_nr) then
   R := R ∪ Update(E, F, s_new!.F)                            // Case 2.
end if
if s_nl ∈ Previous(E, First(F)) ∧ s_nr ∈ Follow(E, Last(F)) then
   R := R ∪ Update(E, G, G + s_new!)                          // Case 3.
end if
return R
```

$w' = and$, onde $F = b_2$, $G = c_3$, $s_{nl} = 1$ e $s_{nr} = 4$. A nova expressão regular obtida é $E' = a_1.(b_2 + c_3 + n_5!)?.d_4$.

Definição 4.1.5 $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ é o conjunto das expressões regulares obtidas pela evolução da sub-expressão $F.G$ de acordo com as posições s_{nl} e s_{nr}, com E sendo a expressão regular original e s_{new} a nova posição correspondente ao símbolo a ser inserido em E.

$evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ é computado pelo Algoritmo 4:

Algorithm 4 $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                    // Solutions calculated so far.
if s_nl ∈ Last(F) ∧ s_nr ∈ First(G) then
   R := R ∪ Update(E, F, F.s_new!)                            // Case 1.
end if
if s_nr ∈ Last(F) ∧ s_nl ∈ First(G) then
   R := R ∪ Update(E, F, s_new!.F) ∪ Update(E, G, G.s_new!)   // Case 2.
end if
return R
```

No Algoritmo 4 são propostas alterações na sub-expressão $F.G$. No primeiro caso, $s_{nl} \in Last(F)$ e $s_{nr} \in First(G)$, a expressão F é substituída por $F.s_{new}!$. No segundo caso, $s_{nr} \in Last(F)$ e $s_{nl} \in First(G)$, duas soluções são propostas: substituíndo F por $s_{new}!.F$ e G por $G.s_{new}!$. Por exemplo, seja $E = (a_1.b_2)^*$, $w = abab$ e $w' = abnab$, onde $F = a_1$, $G = b_2$, $s_{nl} = 2$ e $s_{nr} = 1$, as novas expressões propostas são $E'_1 = (n_3!.a_1.b_2)^*$ e $E'_2 = (a_1.b_2.n_3!)^*$.

Definição 4.1.6 *:* $evClosure(F, E, s_{nl}, s_{nr}, s_{new})$ é o conjunto das expressões regulares obtidas pela evolução das expressões regulares F^+ ou F^* de acordo com as posições s_{nl} e

s_{nr}, onde E é a expressão regular original e s_{new} a posição correspondente ao novo símbolo a ser inserido em E.

evClosure$(E, F, s_{nl}, s_{nr}, s_{new})$ é calculado pelo Algoritmo 5.

Algorithm 5 *evClosure*$(F, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                    // Solutions calculated so far.
if s_nl ∈ Last(F) ∧ s_nr ∈ First(F) then
   R := R ∪ Update(E, F, s_new!.F) ∪ Update(E, F, F.s_new!)   // Case 1.
end if
if s_nr ∈ First(F) ∧ s_nl ∈ Previous(E, s_nr) then
   R := R ∪ Update(E, F, s_new!.F) ∪ Update(E, F, s_new + F)  // Case 2.
end if
if s_nl ∈ First(F) ∧ s_nr ∈ Follow(E, s_nl) then
   R := R ∪ Update(E, F, F.s_new!) ∪ Update(E, F, s_new + F)  // Case 3.
end if
return R
```

No Algoritimo 5 são propostas alterações nas sub-expressões F^* e F^+. No primeiro caso, se $s_{nl} \in Last(F)$ e $s_{nr} \in First(F)$, duas soluções são propostas, substituindo F por $s_{new}!.F$ e $F.s_{new}!$. Por exemplo, seja $E = a_1^*$, $w = aa$ e $w' = ana$, onde $F = a_1$, $s_{nl} = 1$ e $s_{nr} = 1$, as expressões obtidas são $E_1' = (n_2!.a_1)^*$ e $E_2' = (a_1.n_2!)^*$. O segundo caso testa se $s_{nl} \in Previous(E, s_{nr})$ e $s_{nr} = First(F)$, caso o teste seja verdadeiro, duas soluções são propostas, substituindo F por $s_{new}!.F$ e $s_{new}+F$. Por exemplo, considerando $E = a_1.b_2^*$, $w = abb$ e $w' = anbb$, onde $F = b_2$, $s_{nl} = 1$ e $s_{nr} = 2$, as novas expressões regulares propostas são $E_1' = a_1.(n_3!.b_2)^*$ e $E_2' = a_1.(n_3 + b_2)^*$. O terceiro caso é simétrico ao segundo, por exemplo, seja $E = a_1^*.b_2$, $w = aab$ e $w' = aanb$, onde $F = a_1$, $s_{nl} = 1$ e $s_{nr} = 2$, as novas expressões regulares obtidas são $E_1' = (a_1.n_3!)^*.b_2$ e $E_2' = (n_3 + a_1)^*.b_2$.

Definição 4.1.7 *: evOpt*$(F, E, s_{nl}, s_{nr}, s_{new})$ é o conjunto das expressões regulares obtidas pela evolução da expressão regular $F?$, onde E é a expressão regular original e s_{new} é a posição correspondente ao símbolo a ser inserido em E.

evOpt$(E, F, s_{nl}, s_{nr}, s_{new})$ pode é computado pelo algoritmo 6.

Algorithm 6 *evOpt*$(F, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                    // Solutions calculated so far.
if s_nl ∈ Previous(E, First(F)) ∧ s_nr ∈ Follow(E, Last(F)) then
   R := R ∪ Update(E, F, s_new!.F) ∪ Update(E, F, F.s_new!) ∪ Update(E, F, s_new! + F)
end if
return R
```

O Algoritimo 6 propõe alterações na sub-expressão $F?$. Se $s_{nl} \in Previous(E, First(F))$ e $s_{nr} \in Follow(E, Last(F))$ então três novas expressões regulares são propostas. Por exemplo, seja $E = a_1.b_2?.c_3$, $w = ac$ e $w' = anc$, onde $F = b_2$, $s_{nl} = 2$ e $s_{nr} = 3$, as novas expressões geradas são $E'_1 = a_1.(n_4!.b_2)?.c_3$, $E'_2 = a_1.(b_2.n_4!)?.c_3$ e $E'_3 = a_1.(n_4! + b_2)?.c_3$.

4.2 Comparação entre `dGREC` e `GREC`

Como dito na seção anterior, `dGREC` é uma versão do algoritmo `GREC` que não usa o autômato de Glushkov como estrutura de dados principal. A seguir, uma proposição mostra que os resultados obtidos por ambas as abordagens são os mesmos, e uma discussão sobre as diferenças e similaridades é apresentada.

Proposição 4.2.1 *Seja E uma expressão regular e $L(E)$ a linguagem que ela representa. Seja $w[0:n] \in L(E)$. Considerando i como uma alteração (inserção ou exclusão) de uma posição em w ($0 \leq i \leq n$), supondo a exclusão do símbolo $w[i]$, resultando na palavra $w_{del}[0 : n-1]$; ou a inserção de um símbolo na posição i de w, resultando na palavra $w_{ins}[0 : n+1]$. Definindo $s_{nl} = i-1$ e $s_{nr} = i+1$ de acordo com a posição de alteração i de w. Seja $w' \notin L(E)$ a palavra alterada. Seja s_{new} uma posição tal que, $s_{new} \notin Pos(E)$. **dGREC**$(E, E, s_{nl}, s_{nr}, s_{new})$, computado como na Definição 4.1.3, é um conjunto não vazio de expressões regulares, tais que, para cada $E' \in$ **dGREC**$(E, E, s_{nl}, s_{nr}, s_{new})$; nós temos $L(E) \cup \{w'\} \subseteq L(E')$. Mais ainda, o algoritmo que computa **dGREC** é equivalente a **GREC**: a partir dos mesmos parâmetros de entrada, a computação de **dGREC** gera as mesmas expressões regulares obtidas pelo algoritmo **GREC**.*

Prova (esboço): A prova é feita comparando `dGREC` com a versão revisada de `GREC` em [Alv97]. Nesta versão revisada as condições de `GREC` envolvem os conjuntos $Foll$ e $Prev$ (computados sobre o grafo de Glushkov) em vez dos conjuntos Q^+ e Q^- (computados sobre os grafos sem ciclos). A equivalência de `GREC` e `dGREC` é provada em dois principais passos: (1) Cada posição x na expressão regular E refere-se a um estado s no autômato de Glushkov M_E. Então, nós provamos que para cada estado $s \in M_E$, os conjuntos $Foll(s)$ e $Prev(s)$ são equivalentes ao resultado obtido por $Follow(E, x)$ e $Previous(E, x)$,

respectivamente. (2) Nós provamos que cada caso tratado na computação de `dGREC` é também considerado em `GREC` e, para cada um, as mesmas soluções são propostas. A prova da correção de `dGREC` é similar à correção em [Dua05]. □

Em termos de complexidade, como apresentado em [Dua05], `GREC` executa em tempo $O(n^2 + (n^2 \times k))$, onde k é o número de grafos a serem reduzidos e n é o número de nós de cada grafo. Certamente, cada grafo modificado é transformado em uma expressão regular no tempo $O(n^2)$ e `GREC` propõe $k \geq 1$ grafos. O processo original de redução do grafo custa $O(n^2)$.

Considerando agora a complexidade de `dGREC`. Cada expressão regular é dividida em no máximo duas sub-expressões, as quais são computadas recursivamente. A computação do conjunto `dGREC` para cada expressão E necessita de $(n+m)$ passos recursivos, onde n é o número de posições em E e m o número de operadores. Entretanto, para gerar as novas expressões regulares, a função *Update* é usada e, como dito anteriormente, ela realiza a busca de uma sub-expressão de E para realizar a troca. *Update* pode ser implementada recursivamente e, neste caso, sua complexidade é $O(n + m)$ e assim, a complexidade de `dGREC` é $O(k \times (n + m)^2)$ onde k é o número de vezes que *Update* é utilizada.

Uma otimização na implementação de *Update* melhoraria o tempo de execução de `dGREC`. Em vez de executá-la como uma rotina recursiva, pode ser utilizada uma estrutura indexada de posições para cada expressão regular. Considerando esta otimização, a complexidade da implementação de `dGREC` passa a ficar próxima de $O(k \times (n + m))$.

4.3 Trabalhando com mais de uma modificação no processo de evolução das expressões regulares

Os algoritmos GREC e `dGREC` trabalham com apenas uma modificação na palavra original. Dada uma expressão regular E, uma palavra $w \in L(E)$, `GREC` e `dGREC` somente aceitam que seja incluído ou excluído um símbolo na palavra w para gerar a nova palavra w'.

Em [BDAM07] é proposta uma extensão do algoritmo `GREC`, chamada GREC-e, para

trabalhar com mais de uma modificação nas palavras. Baseada nesta proposta, algoritmo `dGREC` foi estendido para aceitar mais de uma modificação na palavra original, o qual chamamos de `dGREC-e`.

Dada uma expressão regular $E = a_1, b_2^*, c_3, d_4$, e a palavra $w = abcd$, considerando agora a possibilidade de mais de uma alteração em w, podemos ter $w' = aefbcdg$. A palavra w' possui a inserção de três símbolos com relação à w. Como os símbolos ef estão juntos, eles podem ser tratados no algoritmo como um único símbolo. E o símbolo g está isolado com relação aos outros símbolos ef.

A idéia do algoritmo `dGREC-e` é encontrar alterações isoladas na palavra alterada com relação à original. Quando existem alterações (inserções de símbolos), onde os símbolos estão um ao lado do outro, estes símbolos são tratados como um único símbolo no algoritmo.

A notação $\$(\alpha_{x_1}, \ldots, \alpha_{x_{n-1}})$ é utilizada para representar estas alterações, cujos símbolos estão um ao lado do outro. Dada a palavra $w = abcd$, e a palavra modificada $w' = aefbcdg$, aplicando a notação $\$(\alpha_{x_1}, \ldots, \alpha_{x_{n-1}})$ em w' temos $w' = a\$(e, f)bcd\$(g)$.

O algoritmo `dGREC-e` é computado igual ao algoritmo `dGREC`, com a adição de uma regra para tratar a notação $\$(\alpha_{x_1}, \ldots, \alpha_{x_{n-1}})$. O comportamento desta regra é igual ao tratamento de um símbolo normal.

Definição 4.3.1 *:* `dGREC-e`$(S, E, s_{nl}, s_{nr}, s_{new})$ é o conjunto de expressões regulares E' obtido pela evolução da expressão regular E. Cada expressão E' é obtida a partir de E, pela inserção de uma nova posição s_{new}, de acordo com as posições s_{nl} e s_{nr}. S é uma expressão regular.

`dGREC-e`$(S, E, s_{nl}, s_{nr}, s_{new})$ é computado da seguinte maneira:

$$
\begin{aligned}
\texttt{dGREC-e}(\emptyset, E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC-e}(\varepsilon, E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC-e}(\alpha_x, E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC-e}(\$(\{\alpha_{x_1}, \ldots, \alpha_{x_{n-1}}\}), E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC-e}(F + G, E, s_{nl}, s_{nr}, s_{new}) &= evOr(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(G, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC-e}(F.G, E, s_{nl}, s_{nr}, s_{new}) &= evAnd(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(G, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC-e}(F^+, E, s_{nl}, s_{nr}, s_{new}) &= evClosure(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(F, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC-e}(F^*, E, s_{nl}, s_{nr}, s_{new}) &= evClosure(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(F, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC-e}(F?, E, s_{nl}, s_{nr}, s_{new}) &= evOpt(F, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC-e}(F, E, s_{nl}, s_{nr}, s_{new})
\end{aligned}
$$

As funções *evOr*, *evAnd*, *evClosure* e *evOpt* ficam com o mesmo comportamente, não sendo necessária alterações. Já nas definições *Null*, *First*, *Last*, *Follow* e *Previous*, o comportamento para a notação $\$(\alpha_{x_1}, \ldots, \alpha_{x_{n-1}})$ é o mesmo dos símbolos.

Ao final da execução de `dGREC-e`, todas as ocorrências de $\$(\alpha_{x_1}, \ldots, \alpha_{x_{n-1}})$ nas expressões regulares geradas, são substituídas da seguinte maneira: $\{(w_0!, \ldots, w_{n-1}!)$, $(w_0|\ldots|w_{n-1})^*,\ (w_0, \ldots, w_{n-1})!\}$. Ou seja, são geradas as combinações possíveis de expressões regulares com os símbolos pertencentes à $\$(\ldots)$.

Dada a expressão regular $E = a, b, \$(c, d)$, aplicando a substituição das ocorrências de $\$(\alpha_{x_1}, \ldots, \alpha_{x_{n-1}})$ em E, obtemos as novas expressões regulares $E'_1 = a, b, c!, d!$, $E'_2 = a, b, (c + d)^*$ e $E'_3 = a, b, (c, d)!$.

O algoritmo *Evolution* é estendido para aceitar várias modificações nas palavras da seguinte maneira:

Definição 4.3.2 *: Evolution-e*(E, w) é o conjunto das expressões regulares (obtidas pela evolução da expressão E) que reconhecem as palavras $w \notin L(E)$ e $w \in L(Evolution\text{-}e(E, w))$

Evolution-e(E, w) é definida pelo Algoritmo 7.

Algorithm 7 *Evolution-e*(E, w)

```
function Evolution-e(E, w)
  var R := ∅                                              // Solutions calculated so far.
  var C := {p | p ∈ First(E) and w_0 = symb(p)}            // Possible current positions.
  var i := 0                                              // Current symbol in w.
  var C' ;                                                // Next possible positions.
  for i := 0 to |w| do
    C' := {p' | p' ∈ Follow(E,p) ∧ w_i = symb(p) ∧ w_{i+1} = symb(p') ∧ p ∈ C}
    if C' = ∅ then
      s_nl := C'
      s_nr := Follow(E, s_nl)
      while (C'' := {p'' | p'' ∈ Follow(E,p') ∧ w_i = symb(p') ∧ w_{i+1} = symb(p'') ∧ p' ∈ C'}) = ∅
      do
        s_new := s_new ∪ NewPos()
        s_nr := C''
        i := i + 1
        C' := C''
      end while
      if R = ∅ then
        R := R ∪ dGREC-e(E, E, s_nl, s_nr, $(s_new))
      else
        for each r ∈ R do
          R := R ∪ dGREC-e(r, r, s_nl, s_nr, $(s_new))
        end for
      end if
    end if
    C := C'
  end for
  return R
end function
```

4.4 Um exemplo detalhado da execução de dGREC

Dada a expressão regular $E = a.(b + c)?.d^{+}.\#$ ($\overline{E} = a_1.(b_2 + c_3)?.d_4^{+}.\#_5$). Para simplificar, usaremos a expressão regular $E = 1.(2 + 3)?.4^{+}.5$. Sejam as palavras $w = abd \in L(E)$ e $w' = abnd \notin L(E)$ com a inserção do símbolo n in w.

Considerando a execução de $Evol(E, w')$, novas expressões regulares E' serão geradas, onde $w \in L(E) \subseteq L(E')$ e $w' \in L(E')$. Em *Evol*, são obtidas as posições $s_{nl} = 2$ e $s_{nr} = 4$, que estão entre os símbolos correspondentes a n na expressão regular E. Vamos considerar que o novo símbolo n será representado pela posição 6, assim $n = 6$. A seguir é apresentado a execução detalhada de dGREC$(E, E, s_{nl}, s_{nr}, n)$, presente no algoritmo *Evol*:

1. Considerando a execução de dGREC$(E, E, s_{nl}, s_{nr}, n)$, com $S = E$, nós temos $S = F.G$

com $F = 1$ e $G = (2+3)?.4^{+}.5$, então $evAnd(F, G, E, s_{nl}, s_{nr}, n)$ é executado. Na execução de *evAnd*, dois testes são executados:

(a) **caso 1:** se $s_{nl} \in Last(F) \ \wedge \ s_{nr} \in First(G) \parallel$ se $2 \in \{1\} \ \wedge \ 4 \in \{2,3,4\}$, como esta condição é falsa, não faz nada.

(b) **caso 2:** se $s_{nr} \in Last(F) \ \wedge \ s_{nl} \in First(G) \parallel$ se $4 \in \{1\} \ \wedge \ 2 \in \{2,3,4\}$, como esta condição é falsa, não faz nada.

Depois da execução acima, as seguintes chamadas são executadas:
(i) $\texttt{dGREC}(1, E, s_{nl}, s_{nr}, n)$ e (ii) $\texttt{dGREC}(\ (2+3)?.4^{+}.5, E, s_{nl}, s_{nr}, n)$.

2. Na execução de (i) $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, n)$ com $S = 1$, como S é um símbolo, dGREC não faz nada.

3. Na execução de (ii) $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, n)$ com $S = (2+3)?.4^{+}.5$ nós temos $S = F.G$ com $F = (2+3)?$ e $G = 4^{+}.5$. Aqui, $evAnd(F, G, E, s_{nl}, s_{nr})$ é executado, onde dois testes são executados:

   (a) **caso 1:** se $s_{nl} \in Last(F) \wedge s_{nr} \in First(G) \parallel$ se $2 \in \{2,3\} \wedge 4 \in \{4\}$, como esta condição é satisfeita, uma nova expressão regular E' é gerada, substituíndo F por $F.5!$, assim $E' = 1.(2+3).6!.4^{+}.5$.

   (b) **caso 2:** se $s_{nr} \in Last(F) \ \wedge \ s_{nl} \in First(G) \parallel$ se $4 \in \{2,3\} \ \wedge \ 2 \in \{4\}$, como esta condição não é satisfeita, então não faz nada.

   Depois da execução acima, dGREC é recursivamente chamado da seguinte maneira:
   (iii) $\texttt{dGREC}(\ (2+3)?, E, s_{nl}, s_{nr}, n)$ e (iv) $\texttt{dGREC}(\ 4^{+}.5, E, s_{nl}, s_{nr}, n)$.

4. Na execução de (iii) $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, n)$ com $S = (2+3)?$, nós temos $S = F?$ com $F = (2+3)$. Então $evOpt(F, E, s_{nl}, s_{nr})$ executada o seguinte teste: se $s_{nl} \in Previous(E, First(F)) \ \wedge \ s_{nr} \in Follow(E, Last(F)) \parallel$ se $2 \in \{1\} \ \wedge \ 4 \in \{4\}$. Como este teste resulta em falso, não faz nada. Agora (v) $\texttt{dGREC}(\ 2+3, E, s_{nl}, s_{nr}, n)$ é recursivamente chamado.

5. Na execução de (iv) $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, n)$ com $S = 4^{+}.5$, nós temos $S = F.G$. Agora, $evAnd(F, G, E, s_{nl}, s_{nr}, n)$ é chamado e os seguintes testes são realizados:

   (a) **caso 1:** se $s_{nl} \in Last(F) \ \wedge \ s_{nr} \in First(G) \parallel$ se $2 \in \{5\} \ \wedge \ 4 \in \{4\}$, como é falsa não faz nada.

   (b) **caso 2:** se $s_{nr} \in Last(F) \ \wedge \ s_{nl} \in First(G) \parallel$ se $4 \in \{5\} \ \wedge \ 2 \in \{4\}$, como é falsa não faz nada.

   Agora, considerando a execução de:
   (vi) $\texttt{dGREC}(\ 4^{+}, E, s_{nl}, s_{nr}, n)$ e (vii) $\texttt{dGREC}(5, E, s_{nl}, s_{nr}, n)$. Na execução de (vii) nós temos $S = 5$, e como S é um símbolo, dGREC não faz nada.

6. Considerando agora a execução de (v) $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, n)$, com $S = 2 + 3$, nós temos $S = F + G$, com $F = 2$ e $G = 3$. Agora, $evOr(F, G, E, s_{nl}, s_{nr})$ é chamado, onde três casos são testados:

   (a) **caso 1:** se $s_{nl} \in Last(F) \ \wedge \ s_{nr} \in Follow(E, s_{nl}) \parallel 2 \in \{2\} \ \wedge \ 4 \in \{4\}$, como a condição é satisfeita, a nova expressão regular E' é obtida substituindo F por $F.n!$, assim $E' = 1.(2.6! + 3).4^{+}.5$.

   (b) **caso 2:** se $s_{nr} \in First(F) \ \wedge \ s_{nl} \in Previous(E, s_{nr}) \parallel$ se $4 \in \{2\} \ \wedge \ 2 \in \{1, 2, 3, 4\}$, como a condição não é satisfeita, não faz nada.

   (c) **caso 3:** se $s_{nl} \in Previous(E, First(F)) \ \wedge \ s_{nr} \in Follow(E, Last(F)) \parallel$ if $2 \in \{1\} \ \wedge \ 4 \in \{4\}$, como a condição é falsa, não faz nada.

   Como a execução de $\texttt{dGREC}(2, E, s_{nl}, s_{nr}, n)$ e $\texttt{dGREC}(3, E, s_{nl}, s_{nr}, n)$, resulta em S igual a um símbolo, em ambos os casos dGREC não faz nada.

7. Por fim, na execução de (vi) $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, n)$, com $S = 4^{+}$, nós temos em S a expressão F^{+}, com $F = 4$. Então $evClosure(F, E, s_{nl}, s_{nr})$ é executado e três casos são testados:

   (a) **caso 1:** se $s_{nl} = F \ \wedge \ s_{nr} = F \parallel$ se $2 = 4 \ \wedge \ 4 = 4$, como esta condição é falsa, não faz nada.

(b) **caso 2:** se $s_{nl} \in \mathit{Previous}(E, s_{nr}) \ \wedge \ s_{nr} = F \parallel$ se $2 \in \{1,2,3,4\} \ \wedge \ 4 = 4$, neste caso a condição é satisfeita, então duas soluções são propostas: a primeira substituíndo F por $n!.F$, e a segunda substituíndo F por $n + F$, assim temos, $E'_1 = 1.(2+3).(6!.4)^+.5$ e $E'_2 = 1.(2+3).(6+4)^+.5$, respectivamente.

(c) **caso 3:** se $s_{nl} = F \ \wedge \ s_{nr} \in \mathit{Follow}(E, s_{nl}) \parallel$ if $2 = 4 \ \wedge \ 4 \in \{4\}$, como a condição não é satisfeita então não faz nada.

Como a execução de `dGREC`$(4, E, s_{nl}, s_{nr}, n)$, gera um símbolo S, `dGREC` não faz nada.

8. `dGREC`$(E, E, s_{nl}, s_{nr}, n)$ retorna recursivamente o conjunto com a união de todas as expressões regulares geradas.

Considerando a execução acima de `dGREC`$(S, E, s_{nl}, s_{nr}, n)$ com $S = E = 1.(2+3)?.4^+.5$, $s_{nl} = 2$, $s_{nr} = 4$ e $n = 6$ o seguinte conjunto de expressões regulares é gerado: $E' = \{1.(2+3).6!.4^+.5,\ 1.(2.6!+3).4^+.5,\ 1.(2+3).(6!.4)^+.5,\ 1.(2+3).(6+4)^+.5\}$. Expandindo o símbolo ! e substituíndo as posições por seus símbolos correspondentes em todas as expressões regulares de E', nós obtemos: $E' = \{a.(b+c).n?.d^+.\#,\ a.(b+c).n^*.d^+.\#,\ a.(b.n?+c).d^+.\#,\ a.(b.n^*+c).d^+.\#,\ a.(b+c).(n?.d)^+.\#,\ a.(b+c).(n^*.d)^+.\#,\ a.(b+c).(n+d)^+.\#\}$. Podemos dizer que $L(E) \subseteq L(E')$ e E' reconhece todas as palavras reconhecidas por E, e também a nova palavra w'.

O algoritmo `dGREC` pode ser utilizado para evoluir expressões regulares, onde seja necessária a manutenção da generalidade com as expressões regulares originais. Ele é mais simples do que o algoritmo `GREC` e gera os mesmos resultados. Pelo fato de trabalhar diretamente na expressão regular, dispensando a transformação da mesma em autômatos finitos de Glushokv, seu desempenho é melhor que o `GREC`.

Detalhes sobre como o algoritmo `dGREC` foi implementado, e alguns testes de desempenho podem ser encontrados no Capítulo 8.

CAPÍTULO 5

ESTUDO DE CASO

Neste capítulo é apresentado um estudo de caso, demonstrando a utilização do algoritmo de evolução de esquemas para XML. Um exemplo da execução do algoritmo irá mostrar passo a passo cada parte do algoritmo. O exemplo irá utilizar o documento XML gerado pela Plataforma Lattes, o qual representa um currículo acadêmico de um professor, aluno, ou pesquisador.

5.1 Plataforma Lattes

A Plataforma Lattes representa uma integração de bases de dados de currículos e de instituições da área de ciência e tecnologia em um único Sistema de Informações, cuja importância se estende, não só às atividades operacionais relacionadas ao CNPq, como também às ações relacionadas a outras agências federais e estaduais, [dDCeT07].

As informações constantes na Plataforma Lattes podem ser utilizadas tanto no apoio a atividades de gestão, como no apoio à formulação de políticas para a área de ciência e tecnologia.

A partir do Currículo Lattes, o CNPq desenvolveu um formato padrão para coleta de informações curriculares de professores, pesquisadores e alunos. O formato escolhido foi XML. Este formato é adotado não só pela Agência, mas também pela maioria das instituições, universidades e institutos de pesquisa do País.

A Plataforma Lattes possui um recurso, no qual é possível exportar o Currículo Lattes para o formato XML. Cada autor pode exportar o seu currículo, e instituições conveniadas ao CNPq podem extrair os currículos de seus pesquisadores da plataforma, no mesmo formato XML.

Com o currículo no formato XML, as instituições podem realizar integrações dos dados currículares de seus pesquisadores com suas bases de dados particulares. Softwares de

interação e extração podem ser criados, para realizar esta tarefa de maneira automatizada. Para isto, a Plataforma Lattes disponibiliza um documento DTD, que descreve como são organizadas as informações nos arquivos XML.

Ao longo do tempo, alterações neste esquemas podem ser necessárias. Porém, como instituições podem já ter desenvolvido sistemas agregados ao documento DTD, esta mudança poderá invalidar o funcionamento destes sistemas. Uma alternativa, seria gerar um novo esquema que seja compatível com o anterior, ou seja, que não perca a sua generalidade com o esquema antigo.

Nossa abordagem demonstra-se ideal para este caso, onde o algoritmo de evolução de esquemas pode gerar, de maneira automatizada, novos esquemas que atendam a todas estas necessidades.

A DTD do Currículo Lattes é apresentada, de maneira simplificada, no Exemplo 9. O documento inicia com o elemento raiz *curriculo-vitae*, o qual aceita os seguintos nós como filhos: *dados-gerais*, sendo obrigatório, representa informações gerais sobre o autor do currículo, tais como, endereços, formação acadêmica, atuações profissionais, areas de atuação, idiomas e prêmios ou títulos do autor. Ainda no nó *dados-gerais* os seguintes atributos são aceitos: *numero-identificador* número gerado pela Plataforma Lattes que identifica cada currículo; *data-atualizacao* data da última atualização do currículo; entre outros.

No elemento *dados-complementares* são armazendas as informações sobre formação complementar, participações em bancas e eventos, orientações em andamento e informações adicionais sobre instituições.

Por fim no elemento *formacao-complementar* são representadas as formações complementares do autor, tais como, formação em extensão universitária, MBA, cursos de curta duração, entre outros.

Exemplo 9 (Lattes.dtd) *Documento DTD simplificado*[1]*, que representa as restrições de como devem ser os Curriculos da Plataforma Lattes no formato XML.*

[1]O documento completo do DTD do Lattes, que possui aproximadamente 274 regras de elementos e 233 de atributos, pode ser encontrado no CD que acompanha este documento, arquivo *lattes.dtd*.

```
<!DOCTYPE curriculo-vitae [
<!ELEMENT curriculo-vitae (dados-gerais, producao-bibliografica?,
   producao-tecnica?, outra-producao?, dados-complementares?)>
<!ATTLIST curriculo-vitae
   sistema-origem-xml cdata #REQUIRED
   numero-identificador cdata #IMPLIED
   formato-data-atualizacao NMTOKEN #FIXED "ddmmaaaa"
   data-atualizacao cdata #IMPLIED
   formato-hora-atualizacao NMTOKEN #FIXED "hhmmss"
   hora-atualizacao cdata #IMPLIED
   xmlns:lattes cdata #IMPLIED>
<!ELEMENT dados-gerais (endereco, formacao-academica-titulacao?,
   atuacoes-profissionais?, areas-de-atuacao?, idiomas?, premios-titulos?)>
   ...
<!ELEMENT dados-complementares (formacao-complementar*,
   participacao-em-banca-trabalhos-conclusao?,
   participacao-em-banca-julgadora?, participacao-em-eventos-congressos?,
   orientacoes-em-andamento?, informacoes-adicionais-instituicoes?,
   informacoes-adicionais-cursos?)>
<!ELEMENT formacao-complementar (
   formacao-complementar-de-extensao-universitaria*, mba*,
   formacao-complementar-curso-de-curta-duracao*, outros*)>
   ...
]>
```

Usando o algoritmo de transformação de DTD para autômato de árvores, apresentado na Seção 2.6, para transformar o documento DTD do Exemplo 9, temos o autômato $\mathcal{A}$, apresentado, de maneira simplificada, no Exemplo 10.

Exemplo 10 (Autômato $\mathcal{A}$ correspondente a DTD $\mathcal{D}$) *Autômato de árvore $\mathcal{A}$, simplificado, correspondente ao documento DTD $\mathcal{D}$, representado no Exemplo 9.*

(1) curriculo-vitae, $\{\{q_{sistema\text{-}origem\text{-}xml}\}, \{q_{numero\text{-}identificador}, q_{formato\text{-}data\text{-}atualizacao}, q_{data\text{-}atualizaca}, q_{formato\text{-}hora\text{-}atualizacao}, q_{hora\text{-}atualizacao}, q_{xmlns:lattes}\}\}$, *dados-gerais*, *producao-bibliografica*?, *producao-tecnica*?,*outra-producao*?, *dados-complementares*? $\rightarrow q_{curriculo\text{-}vitae}$

(2) dados-gerais, $\{\{q_{nome\text{-}completo}, q_{nome\text{-}em\text{-}citacoes\text{-}bibliograficas}, q_{nacionalidade}, q_{sexo}, q_{permissao\text{-}de\text{-}divulgacao}\}, \{q_{cpf}, q_{numero\text{-}do\text{-}passaporte}, q_{pais\text{-}de\text{-}nascimento}, q_{uf\text{-}nascimento}, q_{cidade\text{-}nascimento}, q_{formato\text{-}data\text{-}de\text{-}nascimento}, q_{data\text{-}nascimento}, q_{numero\text{-}identidade}, q_{orgao\text{-}emissor}, q_{uf\text{-}orgao\text{-}emissor}, q_{formato\text{-}data\text{-}de\text{-}emissao}, q_{data\text{-}de\text{-}emissao}, q_{nome\text{-}do\text{-}pai}, q_{nome\text{-}da\text{-}mae}, q_{nome\text{-}do\text{-}arquivo\text{-}de\text{-}foto}, q_{outras\text{-}informacoes\text{-}relevantes}\}\}$, *endereco*,*formacao-academica-titulacao*?, *atuacoes-profissionais*?,*areas-de-atuacao*?, *idiomas*?,*premios-titulos*? $\rightarrow q_{dados\text{-}gerais}$

(190) dados-complementares, $\{\emptyset, \emptyset\}$, *formacao-complementar**,*participacao-em-banca-trabalhos-conclusao*?, *participacao-em-banca-julgadora*?,*participacao-em-eventos-congressos*?,

orientacoes-em-andamento?,*informacoes-adicionais-instituicoes*?, *informacoes-adicionais-cursos*? $\rightarrow q_{dados\text{-}complementares}$

(191) formacao-complementar, {∅, ∅},
formacao-complementar-de-extensao-universitaria*,*mba**,
formacao-complementar-curso-de-curta-duracao*,*outros** $\rightarrow q_{formacao\text{-}complementar}$

(1641) data, {∅, ∅}, ∅ $\rightarrow q_{data}$

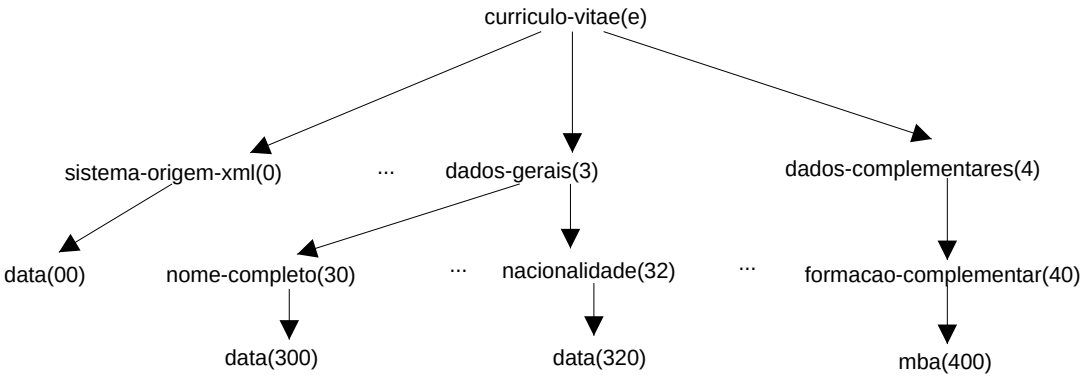
Como exemplo para o estudo de caso, usaremos o currículo no formato de documento XML, demostrado de maneira simplificada no Exemplo 11. O documento possui informações sobre os dados gerais do autor, tais como nome completo, CPF, data de nascimento, entre outros. E dados complementares, onde o autor possui cadastrado um registro sobre um MBA realizado.

Exemplo 11 (curriculo.xml) *Documento XML, simplificado, que representa um Currículo de um autor qualquer, da Plataforma Lattes.*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<curriculo-vitae sistema-origem-xml="lattes_offline"
     data-atualizacao="31102006" hora-atualizacao="095316">
   <dados-gerais nome-completo="robson joao padilha da luz"
      nome-em-citacoes-bibliograficas="luz, r. j. p." nacionalidade="b"
      cpf="00000000000" pais-de-nascimento="brasil" uf-nascimento=""
      cidade-nascimento="" data-nascimento="07121982" sexo="masculino"
      numero-identidade="00000000" orgao-emissor="ssp"
      uf-orgao-emissor="pr" data-de-emissao="18111997"
      numero-do-passaporte="" nome-do-pai="" nome-da-mae=""
      permissao-de-divulgacao="nao">
    ...
   </dados-gerais>
   <dados-complementares>
      <formacao-complementar>
         <mba sequencia-formacao="1" carga-horaria="180"
           status-do-curso="concluido"
           ano-de-inicio="2004" ano-de-conclusao="2005"
           ano-de-obtencao-do-titulo="2005">
         </mba>
      </formacao-complementar>
    ...
   </dados-complementares>
</curriculo-vitae>
```

Para realizar a evolução do autômato $\mathcal{A}$, é necessário que o documento XML seja representado no formato de Árvore XML. No Exemplo 12 é apresentada a árvore XML t correspondente ao documento XML $\mathcal{X}$.

Exemplo 12 (Árvore XML t correspondente ao documento $\mathcal{X}$) *Representação simplificada na forma de árvore XML do documento XML $\mathcal{X}$, demonstrado no Exemplo 11.*



Dada a necessidade de modificações no arquivo XML X representado pela árvore XML t. Iremos representar estas modificações na forma de lista de modificações L, como apresentado na Definição 3.1.2 da Seção 3.1. Considerando as seguintes modificações L:

1. $insert\_element(t, t', pj)$ com $t' = certificacao$ e $pj = 401$ ($p = 40$ *e* $j = 1$). Correspondendo a inserção do elemento $certificacao$ no nó $formacao - complementar$, possibilitando a inclusão de dados de certificações técnicas realizadas pelo autor.

2. $insert\_element(t, t', pj)$ com $t' = curso$ e $pj = 402$ ($p = 40$ *e* $j = 2$). A inserção do elemento $curso$ em *formacao-complementar*, possibilita a inclusão de cursos que o autor tenha feito.

3. $delete\_attribute(t, pj)$ com $pj = 32$ ($p = 3$ *e* $j = 2$), correspondendo ao atributo nacionalidade. Esta alteração possibilita que sejam incluídos currículos sem especificar a nacionalidade.

4. $insert\_attribute(t, att, pj)$ com $att = naturalidade$ e $pj = 318$ ($p = 3$ *e* $j = 18$). A partir desta modificação, é possível adicionar no elemento $dados - gerais$ o atributo naturalidade.

Dada a lista de L modificações realizadas em t apresentada anteriormente, o autômato de árvore $\mathcal{A}$ apresentado no Exemplo 10 e considerando a execução de $ISE(\mathcal{A}, t, L)$, temos a seguir a computação passo a passo de $ISE(\mathcal{A}, t, L)$:

- Inicialmente, $ISE(\mathcal{A}, t, L)$ itera na lista de modificações L com cada elemento da lista sendo chamado de μ.

- Inicialmente temos a modificação número 1 $mu_1 = insert\_element(t, certificacao, 401)$, com $t' = certificacao$ e $pj = 401$ ($p = 40$ *e* $j = 1$).

- A condição if ($type(\mu) = insert\_element(t, t', pj) \wedge \ p \notin inserted\_elements\_pos$) é satisfeita, com relação à modificação mu_1.

- Acompanhando no algoritmo, temos $w = concat\_elements(children(node(p, t)))$, substituíndo, $w = mba$; $w' := w$; o conjunto $insert\_elements$ é criado com todas as modificações $mu_t \in L$, que são do tipo $insert\_element(t, t', p'j')$, e que $p' = p$. Neste caso temos, $insert\_elements = \{insert\_element(t, certificacao, 401)$, $insert\_element(t, curso, 402)\}$.

- A seguir, a palavra w' é alterada de acordo com as modificação presentes em $insert\_elements$. As palavras "certificacao" e "curso" são inseridas em w nas posições 1 e 2 respectivamente. Assim, temos $w' = mba\ certificacao\ curso$.

- Agora temos, $f =$ *formacao-complementar* (nó correspondente à posição p em t).

- A regra $r \in \mathcal{A} =$ [*(191) formacao-complementar,* $\{\emptyset, \emptyset\}$, $E =$ *formacao-complementar-de-extensao-universitaria**, *mba**, *formacao-complementar-curso-de-curta-duracao**, *outros** $\rightarrow q_{formacao\text{-}complementar}$], correspondente ao nó f. Para simplificar a exibição de E, vamos abreviar as palavras *formacao-complementar-de-extensao-universitaria* = *extensao-universitaria* e *formacao-complementar-curso-de-curta-duracao* = *curso-curta-duracao*. Assim temos, $E =$ *extensao-universitaria**, *mba**, *curso-curta-duracao**, *outros**

- Neste caso, será realizada a evolução da expressão regular E, para aceitar a nova palavra w'. Com a execução de $E' = \mathit{Evolution\text{-}e}(E, w')$, temos as seguintes expressões regulares E' geradas:

  1. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, (\mathit{certificacao}|\mathit{curso})^*$
  2. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, (\mathit{certificacao}, \mathit{curso})^*$
  3. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, (\mathit{certificacao}, \mathit{curso})?$
  4. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, \mathit{certificacao}^*, \mathit{curso}^*$
  5. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, \mathit{certificacao}?, \mathit{curso}^*$
  6. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, \mathit{certificacao}^*, \mathit{curso}?$
  7. $\mathit{extensao\text{-}universitaria}^*, \mathit{mba}^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*, \mathit{certificacao}?, \mathit{curso}?$
  8. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, (\mathit{certificacao}|\mathit{curso})^*)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  9. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, (\mathit{certificacao}, \mathit{curso})^*)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  10. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, (\mathit{certificacao}, \mathit{curso})?)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  11. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, \mathit{certificacao}^*, \mathit{curso}^*)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  12. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, \mathit{certificacao}?, \mathit{curso}^*)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  13. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, \mathit{certificacao}^*, \mathit{curso}?)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  14. $\mathit{extensao\text{-}universitaria}^*, (\mathit{mba}, \mathit{certificacao}?, \mathit{curso}?)^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  15. $\mathit{extensao\text{-}universitaria}^*, ((\mathit{certificacao}|\mathit{curso})^*|\mathit{mba})^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  16. $\mathit{extensao\text{-}universitaria}^*, ((\mathit{certificacao}, \mathit{curso})^*|\mathit{mba})^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  17. $\mathit{extensao\text{-}universitaria}^*, ((\mathit{certificacao}, \mathit{curso})?|\mathit{mba})^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$
  18. $\mathit{extensao\text{-}universitaria}^*, (\mathit{certificacao}^*, \mathit{curso}^*|\mathit{mba})^*, \mathit{curso\text{-}de\text{-}curta\text{-}duracao}^*, \mathit{outros}^*$

19. *extensao-universitaria**, (*certificacao*?, *curso**|*mba*)*, *curso-de-curta-duracao**, *outros**
20. *extensao-universitaria**, (*certificacao**, *curso*?|*mba*)*, *curso-de-curta-duracao**, *outros**
21. *extensao-universitaria**, (*certificacao*?, *curso*?|*mba*)*, *curso-de-curta-duracao**, *outros**

- Por fim, a expressão regular E é substuída por E' em $\mathcal{A}$, e a posição $p = 40$ é adicionada no conjunto $inserted\_elements\_pos$.

- Na próxima iteração da lista de modificações L, temos $\mu = insert\_element(t, curso, 402)$, com $p = 40$ e $j = 2$. A primeira condição if $(type(\mu) = insert\_element(t, t', pj) \wedge p \notin inserted\_elements\_pos)$, pois $p \in inserted\_elements\_pos$, ou seja, a modificação μ já foi avaliada na iteração anterior. As outras três condições também não satisfazem neste caso, pois elas tratam dos outros tipos de modificações.

- Seguindo na iteração de L, temos agora $\mu = delete\_attribute(t, 32)$, com $p = 3$ e $j = 2$. Neste caso, a última condição é satisfeita if $(type(\mu) = insert\_attribute(t, att, pj))$.

- Inicialmente, é encontrado o atributo $att = node(pj, t)$ correspondente à posição $pj = 32$, resultando em $att = nacionalidade$. O nó $f = node(p, t)$ é encontrado, correspondente ao nó pai de att, $f =$ *dados-gerais*‘. A regra $r := rule(f, \mathcal{A})$ corresponde à regra 2 que se aplica ao nó f no autômato $\mathcal{A}$. O conjunto s_{op} é acrescido do atributo $q_{nacionalidade}$, que é removido do conjunto s_{comp}.

- Na última iteração em L, temos $\mu = insert\_attribute(t, naturalidade, 308)$ com $p = 3$ e $j = 18$. Neste caso a terceira condição é satisfeita if $(type(\mu) = insert\_attribute(t, att, pj))$. Inicialmente a regra $att - name, \{\emptyset, \emptyset\}, q_{data} \rightarrow q_{att-name}$ é inserida em $\mathcal{A}$. O nó $f = node(3) =$ *dados-gerais* é encontrado, e também a regra 2 $r = rule(f, \mathcal{A})$, correpondente ao nó f. Por fim o conjunto $s_{op} \in r$ é acrescido do elemento $q_{naturalidade}$.

- Ao final da iteração da lista de modificações L o algoritmo retorna o novo autômato de árvore $\mathcal{A}'$.

Seja $\mathcal{X}'$ o documento XML igual ao documento $\mathcal{X}$, do Exemplo 11, com a aplicação da lista de modificações L. A partir da aplicação do algoritmo de evolução incremental de esquemas para XML $ISE(\mathcal{A}, t, L)$, encontramos o novo documento DTD $\mathcal{D}'$ (Exemplo 14), representado pelo autômato de árvore $\mathcal{A}'$ (Exemplo 13), que reconhece tanto do documento XML $\mathcal{X}$, quanto $\mathcal{X}'$. Ele foi evoluído para aceitar as novas necessidades, expressas em $\mathcal{X}'$, e continua aceitando todos os documentos que $\mathcal{D}$, aceitava.

A quantidade de novos documentos DTD gerados pelo algoritmo $ISE(\mathcal{A}, t, L)$ foi aproximadamente 21 documentos. Isto, porque a evolução da expressão regular $E =$ *extensao-universitaria**, *mba**, *curso-curta-duracao**, *outros** resultou em 21 opções, como visto anteriormente. Simulando uma escolha do usuário, foi selecionada a opção de expressão regular número 4 ($E' =$ *extensao-universitaria**, *mba**, *curso-curta-duracao**, *outros**, *certificacao**, *curso**), resultando no autômato de árvore $\mathcal{A}'$.

Exemplo 13 (Autômato de árvore $\mathcal{A}'$ evoluído simplificado) *Autômato de árvore $\mathcal{A}'$ simplificado, resultante do processo evolução do autômato $\mathcal{A}$.*

(1) curriculo-vitae, $\{\{q_{sistema\text{-}origem\text{-}xml}\}, \{q_{numero\text{-}identificador}, q_{formato\text{-}data\text{-}atualizacao}, q_{data\text{-}atualizaca}, q_{formato\text{-}hora\text{-}atualizacao}, q_{hora\text{-}atualizacao}, q_{xmlns:lattes}\}\}$, *dados-gerais*, *producao-bibliografica*?, *producao-tecnica*?,*outra-producao*?, *dados-complementares*? $\rightarrow q_{curriculo\text{-}vitae}$

(2) dados-gerais, $\{\{q_{nome\text{-}completo}, q_{nome\text{-}em\text{-}citacoes\text{-}bibliograficas}, q_{sexo}, q_{permissao\text{-}de\text{-}divulgacao}\}, \{q_{cpf}, q_{numero\text{-}do\text{-}passaporte}, q_{pais\text{-}de\text{-}nascimento}, q_{uf\text{-}nascimento}, q_{cidade\text{-}nascimento}, q_{formato\text{-}data\text{-}de\text{-}nascimento}, q_{data\text{-}nascimento}, q_{numero\text{-}identidade}, q_{orgao\text{-}emissor}, q_{uf\text{-}orgao\text{-}emissor}, q_{formato\text{-}data\text{-}de\text{-}emissao}, q_{data\text{-}de\text{-}emissao}, q_{nome\text{-}do\text{-}pai}, q_{nome\text{-}da\text{-}mae}, q_{nome\text{-}do\text{-}arquivo\text{-}de\text{-}foto}, q_{outras\text{-}informacoes\text{-}relevantes}, q_{\mathbf{nacionalidade}}, q_{\mathbf{naturalidade}}\}\}$, *endereco*,*formacao-academica-titulacao*?, *atuacoes-profissionais*?,*areas-de-atuacao*?, *idiomas*?,*premios-titulos*? $\rightarrow q_{dados\text{-}gerais}$

(190) dados-complementares, $\{\emptyset, \emptyset\}$, *formacao-complementar**,*participacao-em-banca-trabalhos-conclusao*?, *participacao-em-banca-julgadora*?,*participacao-em-eventos-congressos*?, *orientacoes-em-andamento*?,*informacoes-adicionais-instituicoes*?, *informacoes-adicionais-cursos*? $\rightarrow q_{dados\text{-}complementares}$

(191) formacao-complementar, $\{\emptyset, \emptyset\}$, *formacao-complementar-de-extensao-universitaria**,*mba**, *formacao-complementar-curso-de-curta-duracao**,*outros**, *certificacao**,*curso** $\rightarrow q_{formacao\text{-}complementar}$

(1641) data, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{data}$

(1642) certificacao, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{certificacao}$

(1643) curso, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{curso}$

(1644) @naturalidade, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{@naturalidade}$

Aplicando o algoritmo de transformação de autômato de árvore em DTD (Definição 2.6.5, Seção 2.6), no autômato de árvore $\mathcal{A}'$, obtemos o novo documento DTD $\mathcal{D}'$, apresentado no Exemplo 14, que é o novo documento DTD do currículo Lattes, evoluído para aceitar as novas necessidades.

Exemplo 14 (Lattes_novo.dtd) *Documento DTD simplificado que corresponde ao autômato de árvore* $\mathcal{A}'$.

```
<!ELEMENT curriculo-vitae (dados-gerais, producao-bibliografica?,
   producao-tecnica?, outra-producao?, dados-complementares?)>
<!ATTLIST curriculo-vitae
   sistema-origem-xml cdata #REQUIRED
   numero-identificador cdata #IMPLIED
   formato-data-atualizacao NMTOKEN #FIXED "ddmmaaaa"
   data-atualizacao cdata #IMPLIED
   formato-hora-atualizacao NMTOKEN #FIXED "hhmmss"
   hora-atualizacao cdata #IMPLIED
   xmlns:lattes cdata #IMPLIED>
<!ELEMENT dados-gerais (endereco, formacao-academica-titulacao?,
   atuacoes-profissionais?, areas-de-atuacao?, idiomas?, premios-titulos?)>
<!ATTLIST dados-gerais
   nome-completo cdata #REQUIRED
   nome-em-citacoes-bibliograficas cdata #REQUIRED
   nacionalidade cdata
   ...
   naturalidade cdata
>
   ...
<!ELEMENT dados-complementares (formacao-complementar*,
   participacao-em-banca-trabalhos-conclusao?,
   participacao-em-banca-julgadora?, participacao-em-eventos-congressos?,
   orientacoes-em-andamento?, informacoes-adicionais-instituicoes?,
   informacoes-adicionais-cursos?)>
<!ELEMENT formacao-complementar (
   formacao-complementar-de-extensao-universitaria*, mba*,
   formacao-complementar-curso-de-curta-duracao*, outros*,
   certificacao*, curso*)>
   ...
<!ELEMENT certificacao (#PCDATA)>
```

```
<!ELEMENT curso (#PCDATA)>
```

O algoritmo *ISE* demonstrou-se ideal na resolução do problema apresentado neste estudo de caso. A característica de manter a generalidade dos novos esquemas com o esquema original do *ISE*, impede que todos os documentos reconhecidos anteriormente pelo esquema original, não sejam mais reconhecidos pelos esquemas novos. Esta característica é muito importante no caso da Plataforma Lattes, pois existem documentos espalhados por diversas instituições, e alterações no esquema que invalidem estes documentos antigos, podem fazer com que sistemas que utilizam os documentos XML da Plataforma Lattes, não funcionem mais.

O esquema XML (DTD) da Plataforma Lattes é bem extenso, e complexo. Alterações manuais são dificies devido a estas características. Por isto a utilização do algoritmo *ISE* se justifica, pois o mesmo, resolve o problema de forma automática.

CAPÍTULO 6

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho apresentamos uma proposta para trabalhar com o problema de evolução de esquemas para XML de maneira conservativa. Também foi proposto um algoritmo para evoluir expressões regulares (`dGREC`), que é uma extensão do algoritmo `GREC`, apresentado em [BDA+04].

A principal diferença entre `dGREC` e `GREC`, é que o algoritmo `GREC` precisa tranformar a expressão regular em um autômato finito, para então, realizar a evolução da expressão regular. Já o algoritmo `dGREC` trabalha diretamente na expressão regular, resultando na melhoria do tempo de execução do algoritmo.

A maior parte do tempo de execução do algoritmo de evolução incremental de esquemas para XML (*ISE*), é para evoluir as expressões regulares presentes nos esquemas. A melhoria do tempo de complexidade de `dGREC`, resulta em um melhor desempenho do algoritmo *ISE*.

O algoritmo de evolução incremental de esquemas para XML apresentado neste trabalho, pode ser utilizado em diversos ambientes, tais como: em esquemas onde a proliferação de documentos é muito grande, e diversas organizações utilizam estes esquemas, onde desenvolveram sistemas que utilizam este esquema. Mudanças convervativas nestes esquemas, podem evitar que estes sistemas parem de funcionar; Bases de dados em XML, onde existem uma grande quantidade de documentos armazenados e que são estruturados de acordo com algum esquema. Mudanças neste esquema, em um ambiente como este, poderia invalidar todos estes documentos, porém, uma mundança conservativa, manteria a generalidade do novo esquema com todos os documentos antigos, bem como, aceitaria as novas necessidades.

As principais contribuições da nossa proposta sao: (1) Um algoritmo para evoluir expressões regulares de maneira conservativa, chamado `dGREC`, que trabalha diretamente

na expressão regular. (2) Um algoritmo para evoluir esquemas para XML (DTD) de forma conservativa e de maneira automática.

Embora tenhamos diversos benefícios com a abordagem proposta neste trabalho, melhorias podem ser realizadas, como trabalhos futuros, como é citado a seguir:

- Estender o algoritmo `dGREC` para trabalhar com linguagens não regulares. Resultando na possibilidade de usar esquemas não regulares, onde as árvores por ele representadas, podem ter seus nós filhos, restringidos por gramáticas livres de contexto.

- Determinar a complexidade do algoritmo de evolução incremental de esquemas para XML *ISE*.

- Encontrar uma maneira de avaliar as soluções (esquemas) geradas pelo algoritmo *ISE*, visto que o algoritmo pode retornar um quantidade relativamente grande de soluções, dada a quantidade de alterações necessárias. Soluções que medem a distância entre o esquema original e os novos esquemas, gerados pelo algoritmo, podem encontrar um esquema que seja mais parecido com o original, gerando assim, uma solução melhor para o usuário.

- Adaptar o algoritmo para a evolução de esquemas, respeitando outras restrições, alem das restrições de esquema.

- Adaptar o algoritmo de evolução incremental de esquemas para XML para trabalhar com outros tipos de esquemas para XML além do DTD.

- Automatizar a geração da lista de modificações nos arquivos XML. O algoritmo apresentado em [SH04], detecta mudanças entre documentos XML. O mesmo pode ser utilizado para criar esta lista de modificações a partir das mudanças detectadas entre o documento XML original e o modificado.

BIBLIOGRAFIA

[Alv97] Mírian Halfeld Ferrari Alves. Les aspects dynamiques de xml et les services web, 1997. Habilitation a diriger des recherches, Université François Rabelais deTours, em preparação.

[BA03] Béatrice Bouchou and Mírian Halfeld Ferrari Alves. Updates and incremental validation of xml documents. *Proceedings of the 9th International Conference on Data Base Programming Languages*, September 2003.

[BDA+04] Béatrice Bouchou, Denio Duarte, Mírian Halfeld Ferrari Alves, Dominique Laurent, and Martin A. Musicante. Conservative extensions of regular languages. *XXIV International Conference of the Chilean Computer Science Society (SCCC'04)*, 2004. (sccc, pp. 99-109).

[BDAL03] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, and D. Laurent. Extending tree automata to model xml validation under element and attribute constraints. *In ICEIS*, 2003.

[BDAM07] Béatrice Bouchou, Denio Duarte, Mírian Halfeld Ferrari Alves, and Martin A. Musicante. Regular language extension triggered by multiple modifications on words. 2007. Umpublished.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (third edition), February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

[CD00] P. Caron and D.Ziadi. Characterization of glushkov automata. *TCS: Theorical Computer Science*, 2000.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.

[Chi00] Boris Chidlovskii. Using regular tree automata as xml schemas. *Proceedings of the IEEE Advances in Digital Libraries 2000*, May 2000.

[dDCeT07] Conselho Nacional de Desenvolvimento Científico e Tecnológico. Plataforma lattes, April 2007. http://lattes.cnpq.br.

[dLFM07] Robson da Luz, Mírian Halfeld Ferrari, and Martin A. Musicante. Regular expression transformations to extend regular languages (with application to a datalog xml schema validator). *Jornal of Algorithms*, In Press, Corrected Proof, Available online 18 May 2007.

[dLM06] Robson da Luz and Martin Musicante. Regular expression transformations to extend regular languages. In *First Workshop on Logical and Semantic Frameworks, with Applications*, 2006.

[Dua05] Denio Duarte. *Une méthod pour l'évolution de schémas XML préservant la validité des documents*. PhD thesis, Université François-Rabelais de Tours, LI/Campus Blois, 2005.

[GMR05] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. Impact of xml schema evolution on valid documents. *WIDM*, 2005.

[Har01] Elliotte Rusty Harold. *XML Bible*. Hungry Minds, Inc, 2nd edition, 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[Kra00] Diane Kramer. Xem: Xml evolution management. Master's thesis, Worcester Polytechnic Institute, 2000.

[Mar99] Benoît Marchal. *XML by Example*. John Pierce, 1st edition, 1999.

[RHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Html 4.01 specification, December 1999. http://www.w3.org/TR/html401/.

[SH04] Rodrigo C. Santos and Carmem S. Hara. Xkeydiff - um algoritmo semântico para detecção de mudanças entre documentos xml. *REIC (Revista Eletronica de Iniciacao Cientifica)*, 2004.

[Stu03] Tony Stubblebine. *Regular Expression Pocket Reference*. O'Reilly, 1st edition, August 2003.

CAPÍTULO 7

ANEXO II - TESTES DE DESEMPENHO E IMPLEMENTAÇÃO DO ALGORITMO *ISE*

Nesta seção são apresentados alguns resultados obtidos usando o algoritmo *ISE* para evoluir esquemas para XML. Uma versão orientada a objetos do algoritmo *ISE* foi implementada, utilizando a linguagem Java. Diagramas de classes desta implementação são apresentados ao final da seção.

Alguns testes foram realizados para demostrar o desempenho do algoritmo *ISE*. Como *hardware*, foi utilizado um processador CPU AMD Athlon XP 2600+ e 1 GB de memória RAM. E como *software* foram utilizados o sistema operacional Microsoft Windows XP Professional e a máquina virtual do Java *Java HotSpot Client VM version 1.5.0*.

Seja o documento XML $\mathcal{X}$ apresentado no Exemplo 11 (Seção 5), que representa um curriculo da Plataforma Lattes. Seja t a árvore XML correspondente ao documento XML $\mathcal{X}$, apresentada no Exemplo 12 (Seção 5). Dado o autômato de árvore $\mathcal{A}$ correspondente ao documento DTD $\mathcal{D}$, apresentados na Seção 5 nos Exemplos 10 e 9, respectivamente.

Considerando algumas modificações na árvore XML t, o algoritmo *ISE* gera novos autômatos de árvores $\mathcal{A}'$, que aceitam estas novas modificações. Na Tabela 7 são apresentados alguns casos de modificações na árvore t, que foram utilizados nos testes de desempenho do algoritmo *ISE*.

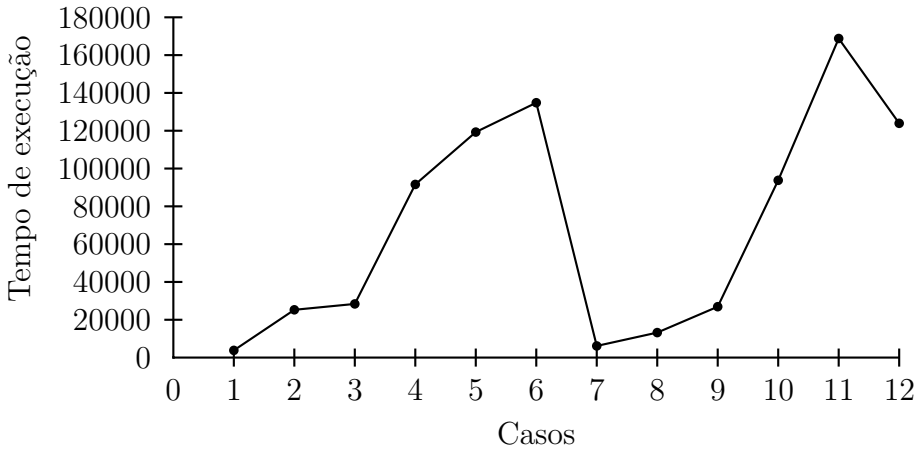
Na Figura 7.1 é apresentado o desempenho obtido pelo algoritmo em cada caso de teste. O tempo está representado em um centésimo de nano segundo ($valor * 100$). Estes resultados foram obtidos através da implementação realizada em Java, usando o paradigma de orientação a objetos, onde os diagramas de classes serão apresentados no final deste capítulo.

Analisando os resultados obtidos, podemos verificar que existe uma relação entre o

Tabela 7.1: Lista de Modificações.

Caso	**Lista de Modificações** L
1	$insert\_attribute(t, rg, 3)$
2	$insert\_element(t, certificacao, 401)$
3	$insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$
4	$insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$ $insert\_attribute(t, rg, 3)$ $insert\_attribute(t, telefone2, 3181)$
5	$insert\_element(t, experiencia - no - exterior, 41)$ $insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$ $insert\_attribute(t, rg, 3)$ $insert\_attribute(t, telefone2, 3181)$
6	$insert\_element(t, experiencia - profissional, 4)$ $insert\_element(t, experiencia - no - exterior, 41)$ $insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$ $insert\_attribute(t, rg, 3)$ $insert\_attribute(t, telefone2, 3181)$
7	$insert\_attribute(t, rg, 3)$ $delete\_attribute(t, 34)$ – atributo nacionalidade
8	$delete\_attribute(t, 34)$ – atributo nacionalidade $delete\_element(t, 318)$ – elemento endereco $insert\_element(t, certificacao, 401)$
9	$delete\_element(t, 3)$ – atributo dados-gerais $insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$
10	$insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$ $insert\_attribute(t, rg, 3)$ $delete\_attribute(t, 34)$ – atributo nacionalidade $insert\_attribute(t, telefone2, 3181)$
11	$insert\_element(t, experiencia - no - exterior, 41)$ $insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$ $insert\_attribute(t, rg, 3)$ $delete\_attribute(t, 34)$ – atributo nacionalidade $insert\_attribute(t, telefone2, 3181)$
12	$insert\_element(t, experiencia - profissional, 4)$ $insert\_element(t, experiencia - no - exterior, 41)$ $insert\_element(t, certificacao, 401)$ $insert\_element(t, curso, 402)$ $insert\_attribute(t, rg, 3)$ $delete\_attribute(t, 34)$ – atributo nacionalidade $insert\_attribute(t, telefone2, 3181)$

Figura 7.1: Gráfico com o desempenho do algoritmo *ISE*



desempenho e a quantidade de modificações necessárias em cada caso. Isto se deve ao fato de que, quanto mais modificações necessárias, menor será o desempenho do algoritmo, pois o algoritmo trabalha em cada modificação individualmente. O tipo de modificação que leva mais tempo para executar é o de inserção de elementos, pois para realizar a evolução do esquema neste caso, é necessário a utilização do algoritmo de evolução de expressões regulares `dGREC`, e nos outros casos, somente alterações simples no esquema são necessárias.

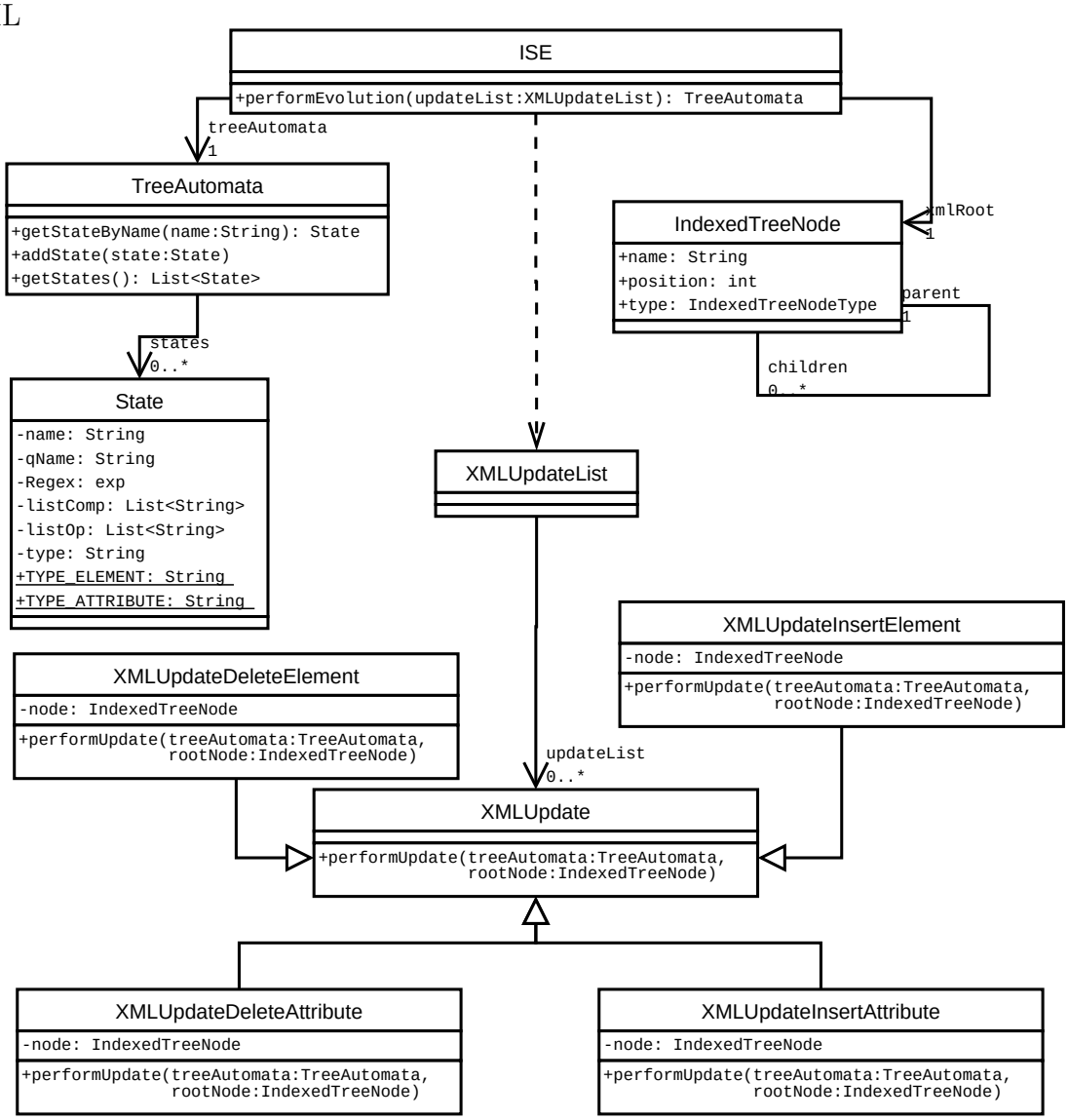
Considerando a implementação orientada a objetos do algoritmo *ISE*. As classes utilizadas nesta implementação podem ser visualizadas na diagrama de classes apresentado na Figura 7.2.

A seguir são apresentadas descrições sobre cada classe do diagrama apresentado na Figura 7.2:

- **Classe TreeAutomata**: Representa um autômato de árvore. Possui uma lista de estados (State).

- **Classe State**: Representa um estado de um autômato de árvore. Possui um nome (*name*), que representa o nome correspondente ao atributo ou elemento XML. O atributo *qName* é composto pelo nome do estado com o prefixo *q_*. A classe ainda possui os atributos *exp* que representa a expressão regular correspondente a este es-

Figura 7.2: Diagrama de Classes Algoritmo de Evolução Incremental de Esquemas para XML



tado, e as listas *listComp* e *listOp* que representam a lista de atributos obrigatórios e opcionais, respectivamente. Os objetos desta classe podem ser de dois tipos (abributo *type*): *Element* ou *Attribute*.

- **Classe IndexedTreeNode**: Representa um nó de uma árvore XML t. Os documentos XML são representados no sistema usando esta classe. A posição no nós em relação ao nó pai (atributo *parent*) é representada pelo atributo *position*. O atributo *name* representa o nome do nó correspondente ao nome do atributo ou elemento, quando o tipo (atributo *type*) for *attribute* ou *element*, respectivamente, ou a constante *DATA* quando o nós representa um nós tipo texto da árvore XML.

Através da relação *parent* → *children* é montada a árvore XML.

- **Classe XMLUpdate**: Representa uma modificação na árvore XML. Podendo ser do tipo: *XMLUpdateInsertElement*, *XMLUpdateInsertAttribute*, *XMLUpdateDeleteElement* ou *XMLUpdateDeleteAttribute*.

- **Classe XMLUpdateInsertElement**: Esta classe representa uma modificação de inserção de elemento na árvore XML ($insert\_element(t, t', pj)$). Ela implementa o método $performUpdate(treeAutomata : TreeAutomata,$
  $rootNode : IndexedTreeNode)$, onde é realizada a evolução do autômato de árvore $treeAutomata$, chamando o algoritmo `dGREC-e` para evoluir a expressão regular E, obtida através do atributo *node* da seguinte maneira:
  $E = treeAutomata.getStateByName(node.parent.name).exp$.

- **Classe XMLUpdateInsertAttribute**: Esta classe representa uma modificação de inserção de atributo na árvore XML ($insert\_attribute(t, att, pj)$). Ela implementa o método $performUpdate(treeAutomata : TreeAutomata,$
  $rootNode : IndexedTreeNode)$, adicionando o atributo com o nome $node.name$ na lista $treeAutomata.getStateByName(node.parent.name).lstOpt$.

- **Classe XMLUpdateDeleteElement**: Esta classe representa uma modificação de exclusão de um elemento na árvore XML ($delete\_element(t, pj)$). Ela implementa o método $performUpdate(treeAutomata : TreeAutomata,$
  $rootNode : IndexedTreeNode)$, onde é realizada a evolução do autômato de árvore $treeAutomata$, chamando o algoritmo *Delete* para evoluir a expressão regular E, obtida através do atributo *node* da seguinte maneira:
  $E = treeAutomata.getStateByName(node.parent.name).exp$.

- **Classe XMLUpdateDeleteAttribute**: Esta classe representa uma modificação de inserção de atributo na árvore XML ($delete\_attribute(t, pj)$). Ela implementa o método $performUpdate(treeAutomata : TreeAutomata,$
  $rootNode : IndexedTreeNode)$, onde é realizada a evolução do autômato de árvore

treeAutomata, adicionando o atributo com o nome *node.name* na lista $treeAutomata.getStateByName(node.parent.name).lstOpt$ e removendo da lista $treeAutomata.getStateByName(node.parent.name).lstComp$.

- **Classe XMLUpdateList**: Representa uma lista de modificações realizadas numa árvore XML.

- **Classe ISE**: Esta classe representa o algoritmo de evolução incremental de esquemas para XML. Ela possui os atributos *treeAutomata* (autômato de árvore) e *xmlRoot* (árvore XML). O método $performEvolution(updateList : XMLUpdateList)$ realiza a evolução do autômato de árvore *treeAutomata* de acordo com a lista de modificações *updateList*, ele retorna um conjunto de autômatos de árvore $treeAutomata'$ que reconhecem todos os documentos reconhecidos por *treeAutomata*, e também os documentos que estão de acordo com a lista de modificações *updateList*.

CAPÍTULO 8

ANEXO I - TESTES DE DESEMPENHO E IMPLEMENTAÇÃO DO ALGORITMO DGREC

Nesta seção são apresentados alguns resultados obtidos usando o algoritmo dGREC para evoluir expressões regulares. Foi implementada uma versão orientada a objetos do algoritmo dGREC, utilizando a linguagem Java. Diagramas de classes são apresentados demonstrando como esta implementação foi realizada.

Alguns testes foram realizados para demostrar a performance do dGREC. O *hardware* utilizado foi um processador CPU AMD Athlon XP 2600+ e 1 GB de memória RAM. E o *software* utilizado foi o sistema operacional *Microsoft Windows XP Professional* e, *Java HotSpot Client VM version 1.5.0* como máquina virtual do Java.

Considerando o problema de evolução de expressões regulares, dada uma expressão regular E, a palavra $w \in L(E)$ e a palavra w', igual à w, com um alteração (inserção do símbolo n em uma posição p de w), onde $w' \notin L(E)$. A implementação de dGREC é executada sobre $Evol(E, w')$. Os testes apresentados na Tabela 8 foram realizados em $Evol(E, w')$.

Para cada caso apresentado Tabela 8, o algoritmo $Evol(E, w')$ gera novas expressões regulares E', onde $w' \in L(E')$, $w \in L(E')$ e $L(E) \subseteq L(E')$. Estes resultados são apresentados na Tabela 8.

Na Figura 8.1 é apresentado o desempenho obtido pelo algoritmo em cada caso de teste. O tempo está representado em nano segundos. Estes resultados foram obtidos através da implementação realizada em Java, usando o paradigma de orientação a objetos, onde os diagramas de classes serão apresentados no final deste capítulo.

Analisando os resultados obtidos, podemos verificar que o algoritmo teve um bom desempenho. Onde existe uma relação entre o desempenho e a quantidade de opções geradas pelo algoritmo em cada caso. Isto se deve ao fato de que, a relação de quantidade

Tabela 8.1: Lista de casos de testes realizados no algoritmo `dGREC`.

	E	w	w'
1	$a.(b+c)?.d$	ad	and
2	$a.(b+c)$	abc	$anbc$
3	$(a.b)^*$	$abab$	$abnab$
4	$(a)^*$	aa	ana
5	$a.b^*$	abb	$anbb$
6	$a^*.b$	aab	$aanb$
7	$a.b?.c$	ac	anc
8	$a.(b+c)?.d^+$	abd	$abnd$
9	$(a.b)^+.c^*$	$ababc$	$abanbc$
10	$(a.b)^*.c.(d+e+f)^+.g?.h$	$ababcddedfeeh$	$ababcdndedfeeh$
11	$(a.b)^*.c.(d+e+f)^+.g?.h$	$ababcddedfeeh$	$abanbcddedfeeh$
12	$(a.b)^*.c.(d+e+f)^+.g?.h$	$ababcddedfeeh$	$ababcddedfeehn$
13	$a^*.b^+.c^+.d^*.e.f?.g$	$bbbcdeg$	$bbbncdeg$
14	$a^*.b^+.c^+.d^*.e.f?.g$	$bbbcdeg$	$bbbcdeng$
15	$(a^*.b)^+.(c.d.e.f)?.g?$	$aaabaabcdef$	$aanabaabcdef$
16	$(a^*.b)^+.(c.d.e.f)?.g?$	$aaabaabcdef$	$aaabnaabcdef$
17	$(a^*.b)^+.(c.d.e.f)?.g?$	$aaabaabcdef$	$aaabaabncdef$
18	$(a^*.b)^+.(c.d.e.f)?.g?$	$aaabaabcdef$	$aaabaabcdnef$
19	$(a^*.b)^+.(c.d.e.f)?.g?$	$aaabaabcdef$	$aaabaabcdefn$

de opções geradas, está diretamente ligada à quantidade de condições que são satisfeitas durante a execução do algoritmo, ou seja, de acordo com a necessidade de mudanças (posições s_{nl}, s_{nr} na expressão regular, utilizadas para as verificações de inserção de um novo símbolo ou palavra s_{new}, Seção 4), mais trechos de código (*evOr*, *evAnd*, *evClosure* e *evOpt*) são executados, gerando assim um tempo maior de execução do algoritmo.

Na implementação orientada a objetos do algoritmo `dGREC` foram utilizadas algumas classes que representam as expressões regulares, bem como operações entre as mesmas. A Figura 8.2 apresenta estas classes, e suas relações.

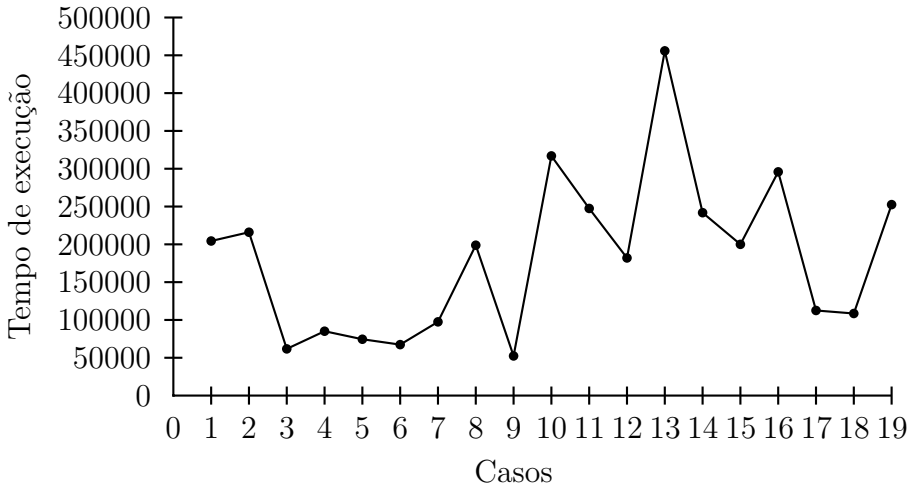
A seguir são apresentadas descrições sobre cada classe do diagrama apresentado na Figura 8.2:

- **Classe Regex**: Representa uma expressão regular. O atributo *pos* representa as posições desta expressão regular.

Tabela 8.2: Resultados obtidos após a execução do algoritmo *Evol* em cada caso da Tabela 8.

	Expressões regulares E' geradas
1	$a.n?.(b+c)?.d$, $a.n^*.(b+c)?.d$, $a.(b+c)?.n?.d$, $a.(b+c)?.n^*.d$, $a.(n?+(b+c)?).d$, $a.(n^*+(b+c)?).d$, $a.(b+c+n?)?.d$, $a.(b+c+n^*)?.d$
2	$a.n?.(b+c).c?$, $a.n^*.(b+c).c?$, $a.n?.(b+c).c^*$, $a.n^*.(b+c).c^*$, $a.n?.(b.c?+c)$, $a.n^*.(b.c?+c)$, $a.n?.(b.c^*+c)$, $a.n^*.(b.c^*+c)$, $a.(n?.b+c).c?$, $a.(n^*.b+c).c?$, $a.(n?.b+c).c^*$, $a.(n^*.b+c).c^*$, $a.(n?.b.c?+c)$, $a.(n^*.b.c?+c)$, $a.(n?.b.c^*+c)$, $a.(n^*.b.c^*+c)$
3	$(a.b.n?)^*$, $(a.b.n^*)^*$, $(n?.a.b)^*$, $(n^*.a.b)^*$
4	$(n?.a)^*$, $(n^*.a)^*$, $(a.n?)^*$, $(a.n^*)^*$, $(n?+a)^*$, $(n^*+a)^*$
5	$a.n?.b^*$, $a.n^*.b^*$, $a.(n?.b)^*$, $a.(n^*.b)^*$, $a.(n?+b)^*$, $a.(n^*+b)^*$
6	$a^*.n?.b$, $a^*.n^*.b$, $(a.n?)^*.b$, $(a.n^*)^*.b$, $(n?+a)^*.b$, $(n^*+a)^*.b$
7	$a.n?.b?.c$, $a.n^*.b?.c$, $a.b?.n?.c$, $a.b?.n^*.c$, $a.(n?+b?).c$, $a.(n^*+b?).c$
8	$a.(b+c)?.n?.d^+$, $a.(b+c)?.n^*.d^+$, $a.(b.n?+c)?.d^+$, $a.(b.n^*+c)?.d^+$, $a.(b+c)?.(n?.d)^+$, $a.(b+c)?.(n^*.d)^+$, $a.(b+c)?.(n?+d)^+$, $a.(b+c)?.(n^*+d)^+$
9	$(a.n?.b)^+.c^*$, $(a.n^*.b)^+.c^*$
10	$(a.b)^*.c.(d.n?+e+f)^+.g?.h$, $(a.b)^*.c.(d.n^*+e+f)^+.g?.h$, $(a.b)^*.c.(n?.d+e+f)^+.g?.h$, $(a.b)^*.c.(n^*.d+e+f)^+.g?.h$, $(a.b)^*.c.(d+e+f+n?)^+.g?.h$, $(a.b)^*.c.(d+e+f+n^*)^+.g?.h$
11	$(a.n?.b)^*.c.(d+e+f)^+.g?.h$, $(a.n^*.b)^*.c.(d+e+f)^+.g?.h$
12	$(a.b)^*.c.(d+e+f)^+.g?.h.n?$, $(a.b)^*.c.(d+e+f)^+.g?.h.n^*$
13	$a^*.b^+.n?.c^+.d^*.e.f?.g$, $a^*.b^+.n^*.c^+.d^*.e.f?.g$, $a^*.(b.n?)^+.c^+.d^*.e.f?.g$, $a^*.(b.n^*)^+.c^+.d^*.e.f?.g$, $a^*.(n?+b)^+.c^+.d^*.e.f?.g$, $a^*.(n^*+b)^+.c^+.d^*.e.f?.g$, $a^*.b^+.(n?.c)^+.d^*.e.f?.g$, $a^*.b^+.(n^*.c)^+.d^*.e.f?.g$, $a^*.b^+.(n?+c)^+.d^*.e.f?.g$, $a^*.b^+.(n^*+c)^+.d^*.e.f?.g$
14	$a^*.b^+.c^+.d^*.e.n?.f?.g$, $a^*.b^+.c^+.d^*.e.n^*.f?.g$, $a^*.b^+.c^+.d^*.e.f?.n?.g$, $a^*.b^+.c^+.d^*.e.f?.n^*.g$, $a^*.b^+.c^+.d^*.e.(n?+f?).g$, $a^*.b^+.c^+.d^*.e.(n^*+f?).g$
15	$((n?.a)^*.b)^+.(c.d.e.f)?.g?$, $((n^*.a)^*.b)^+.(c.d.e.f)?.g?$, $((a.n?)^*.b)^+.(c.d.e.f)?.g?$, $((a.n^*)^*.b)^+.(c.d.e.f)?.g?$, $((n?+a)^*.b)^+.(c.d.e.f)?.g?$, $((n^*+a)^*.b)^+.(c.d.e.f)?.g?$
16	$(a?.b.n?)^+.(c.d.e.f)?.g?$, $(a?.b.n^*)^+.(c.d.e.f)?.g?$, $(n?.a?.b)^+.(c.d.e.f)?.g?$, $(n^*.a?.b)^+.(c.d.e.f)?.g?$, $((n?.a)^*.b)^+.(c.d.e.f)?.g?$, $((n^*.a)^*.b)^+.(c.d.e.f)?.g?$, $((n?+a)^*.b)^+.(c.d.e.f)?.g?$, $((n^*+a)^*.b)^+.(c.d.e.f)?.g?$
17	$(a^*.b)^+.n?.(c.d.e.f)?.g?$, $(a^*.b)^+.n^*.(c.d.e.f)?.g?$
18	$(a^*.b)^+.(c.d.n?.e.f)?.g?$, $(a^*.b)^+.(c.d.n^*.e.f)?.g?$
19	$(a^*.b)^+.(c.d.e.f)?.g?.n?$, $(a^*.b)^+.(c.d.e.f)?.g?.n^*$, $(a^*.b)^+.(c.d.e.f)?.n?.g?$, $(a^*.b)^+.(c.d.e.f)?.n^*.g?$, $(a^*.b)^+.(c.d.e.f)?.(n?+g?)$, $(a^*.b)^+.(c.d.e.f)?.(n^*+g?)$

Figura 8.1: Gráfico com o desempenho do algoritmo `dGREC`



- **Classe WordRegex**: Representa uma expressão regular com somente um símbolo ou palavra, ex: a. O atributo *value* representa a símbolo ou a palavra da expressão, e o atributo *pos* representa a posição que esta palavra está na expressão regular.

- **Classe OptRegex**: Representa uma expressão regular do tipo $E = value?$.

- **Classe AndRegex**: Representa uma expressão regular do tipo $E = value1, value2$.

- **Classe OrRegex**: Representa uma expressão regular do tipo $E = value1|value2$.

- **Classe StarRegex**: Representa uma expressão regular do tipo $E = value^*$.

- **Classe StarOneRegex**: Representa uma expressão regular do tipo $E = value^+$.

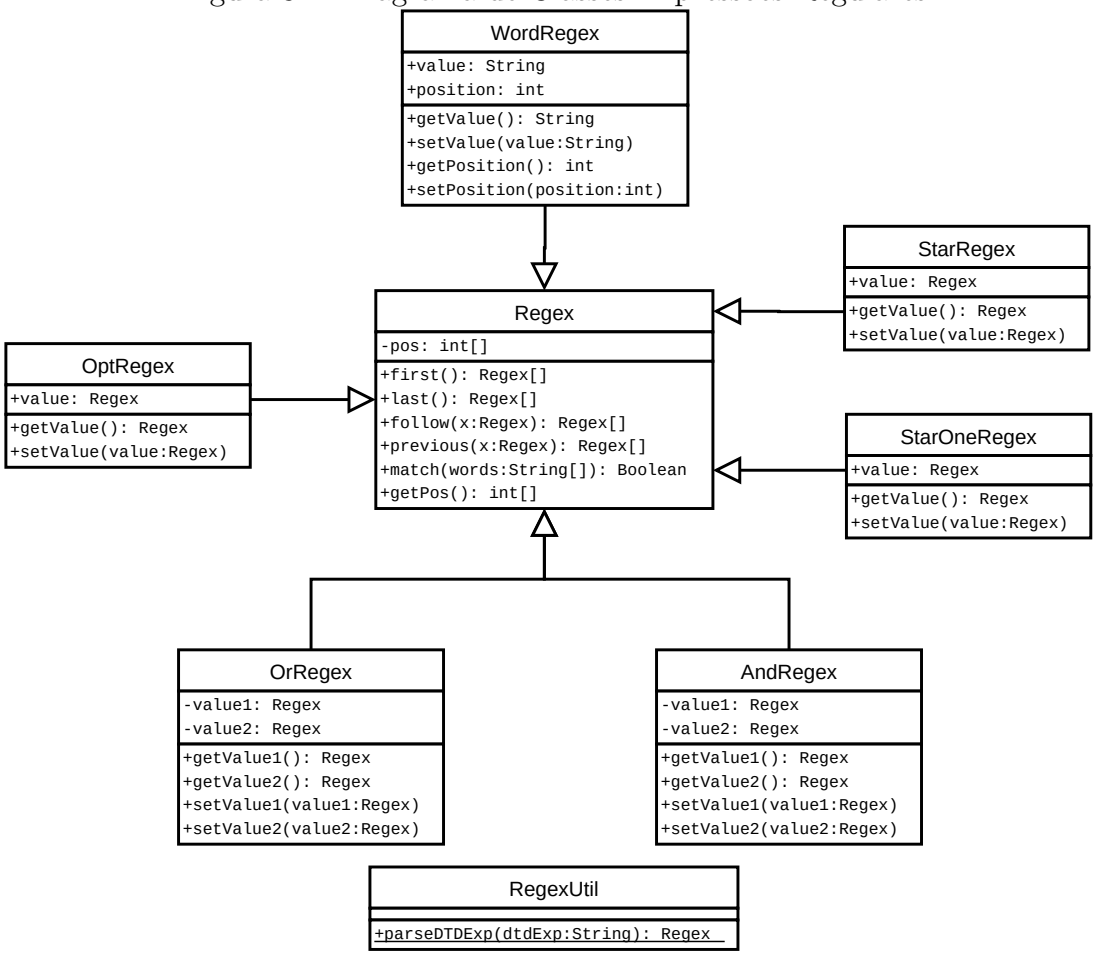
- **Classe RegexUtil**: Possui o método $parseDTDExp(dtdExp : String) : Regex$, utilizado para transformar uma expressão regular no formato da expressão utilizada em documentos DTD para uma instância da classe *Regex*.

Na Figura 8.3 são apresentadas as classes que implementam o algoritmo de evolução de expressões regulares `dGREC`.

A seguir, é apresenta a descrição de cada classe presente no diagrama da Figura 8.3.

- **Classe EvolutionAnd**: Realiza a evolução da expressão regular $regex$ do tipo $AndRegex$, de acordo com as modificações rm ($RegexModification$). Caso gere

Figura 8.2: Diagrama de Classes Expressões Regulares



uma nova expressão regular E', adiciona E' na lista $regexList$.

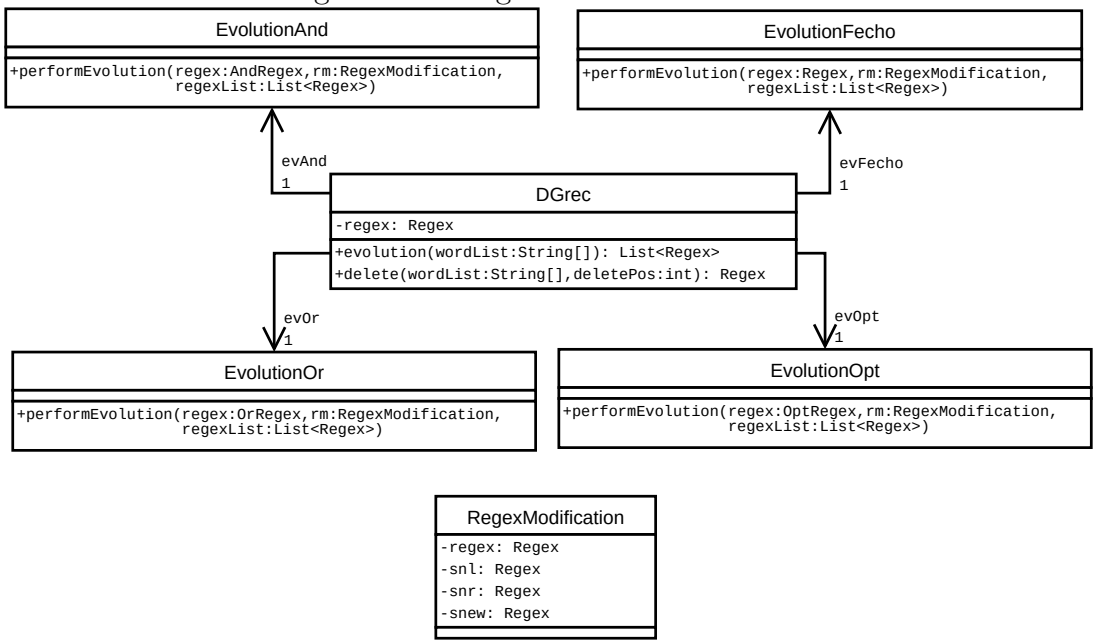
- **Classe EvolutionFecho**: Realiza a evolução da expressão regular $regex$ do tipo $StarRegex$ ou $StarOneRegex$, de acordo com as modificações rm ($RegexModification$). Caso gere uma nova expressão regular E', adiciona E' na lista $regexList$.

- **Classe EvolutionOr**: Realiza a evolução da expressão regular $regex$ do tipo $OrRegex$, de acordo com as modificações rm ($RegexModification$). Caso gere uma nova expressão regular E', adiciona E' na lista $regexList$.

- **Classe EvolutionOpt**: Realiza a evolução da expressão regular $regex$ do tipo $OptRegex$, de acordo com as modificações rm ($RegexModification$). Caso gere uma nova expressão regular E', adiciona E' na lista $regexList$.

- **Classe DGrec**: Implementa o algoritmo de evolução de expressões regulares, utili-

Figura 8.3: Diagrama de Classes `dGREC`



zando as classes *EvolutionAnd, EvolutionFecho, EvolutionOr* e *EvolutionOpt*. Possui o atributo *regex* que representa a expressão regular original a ser modificada. O método $evolution(wordList : String[])$ realiza a evolução da expressão regular *regex*, onde *wordList* é uma palavra, tal que, existe uma lista de palavras $w \in L(regex)$, e *wordList* é contruída com a inserção de uma ou mais palavras em w. Um conjunto de expressões regulares $regex'$ são retornadas pelos método *evolution*, onde $wordList \in L(regex')$, $w \in L(regex')$ e Ł$(regex) \subseteq L(regex')$. O método $delete(wordList : String[], deletePos : int)$ evolui a expressão regular *regex*, tornando a palavra $wordList[deletePos]$ em *regex* opcional.

- **Classe RegexModification**: Representa uma modificação a ser realizada na expressão regular *regex* de acordo com as posições das expressões regulares *snl* e *snr*. O atributo *snew* representa a expressão regular a ser inserida em *regex*, podendo ser somente uma simples palavra, como uma expressão regular mais complexa.